

Cours d'informatique en MPI au lycée Hoche

2023-2024

Le prof et surtout les étudiants scribes



Table des matières

I	Cours	9
0	Jeux	11
0.0	Banalités	11
0.1	Définitions et vocabulaire	11
0.2	Attracteurs	13
0.3	Algorithme Min-Max	15
0.4	Algorithme d'élagage $\alpha\beta$	15
1	Langages	17
1.0	Définitions et vocabulaire	17
1.1	Opérations sur les langages	19
1.2	Question de régularité (ou rationalité)	19
1.2.1	Expressions régulières (ou rationnelles)	19
1.2.2	Langage d'une expression régulière	20
1.2.3	Expression linéaire	21
1.3	Expressions POSIX étendues	21
1.4	Langages locaux	21
1.4.1	Opérations sur les langages locaux (HP)	22
1.4.2	Lien avec les expressions régulières	22
	Exercices	23
2	Graphes	27
2.0	Révisions	27
2.1	Graphes bipartis	29
2.1.1	Couplage dans un graphe biparti	29
2.1.2	Lien aux flots dans les graphes (HP)	32
2.2	Graphes orientés acycliques (DAG)	33
2.2.1	Tri topologique	34
2.2.2	Composantes fortement connexes	35
2.2.3	2-SAT	38
2.3	Arbres couvrants	39
2.4	Distances dans les graphes	41
2.4.1	Définitions	41
2.4.2	DIJKSTRA	42
2.4.3	FLOYD-WARSHALL	43
2.4.4	Retour sur la programmation dynamique	45
3	Concurrence et parallélisme	47
3.0	Banalités	47
3.1	Section critique et exclusion mutuelle	48
3.1.1	Algorithme de PETERSON	49
3.2	Mutex à n processus	50
3.2.1	Généralisation de PETERSON (HP)	50
3.2.2	Boulangerie de LAMPORT	50
3.3	Sémaphores	51

3.4	Implémentations	51
3.4.1	En C	52
3.4.2	En OCaml	52
3.5	Quelques exemples au programme	53
3.5.1	Dîner de philosophes	53
3.5.2	Rendez-vous	54
3.5.3	Producteurs-consommateurs	54
3.5.4	Barrière de synchronisation	54
	Exercices	54
4	Classes de complexité	61
4.0	Banalités autour de SAT (HP)	61
4.0.1	Sudoku	61
4.0.2	Exécuter un programme	62
4.1	Problèmes de décision	62
4.2	Réduction de problèmes	64
4.2.1	Exemples en logique (HP)	65
4.2.2	Exemples dans les graphes (HP)	67
4.3	Autres classes de complexité (HP)	69
4.4	Décidabilité	70
4.4.1	Histoire sympa	70
4.4.2	Définition	71
	Exercices	72
5	Automates	75
5.0	Introduction	75
5.1	Automates finis déterministes	76
5.1.1	Définitions	76
5.1.2	Langage d'automate	77
5.1.3	Lemme de l'étoile	79
5.2	Autour des automates déterministes et études linguistiques	80
5.3	Automates non déterministes	80
5.3.1	Définition	80
5.3.2	Automates à transition spontanée	82
5.4	Opérations sur les langages reconnaissables	83
5.5	Théorème de KLEENE	85
5.5.1	Le théorème	85
5.5.2	Algorithme de BERRY-SETHI	85
	Exercices	85
6	Optimisation et approximation	89
6.0	Introduction	89
6.1	Briques élémentaires et algorithmes Gobelins	90
6.1.1	Définitions	90
6.1.2	L'algorithme glouton	90
6.1.3	Autour du voyageur de commerce	90
6.1.4	Le problème de la vache qui rit (ou du coupon collector)	92
6.2	Algorithmes probabilistes ou algorithmes Orcs	92
6.2.1	Algorithme de KARGER	93
6.2.2	Trions (avec le tri rapide)	94
6.2.3	Mélangeons (Algorithme de KNUTH FISHER-YATES)	96
6.3	Quelques difficultés concernant les probabilités en informatique	96

7	Logique	97
7.0	Banalités	97
7.1	Règles de déduction naturelles	97
7.2	Correction de la déduction naturelle	100
7.3	Dans la vie il y a 42 implications, ou comment se tordre la tête	100
7.4	Quantificateurs (1 ^{er} ordre)	101
	Exercices	102
8	Grammaires	107
8.0	Banalités	107
8.1	Définitions	108
8.2	Lemme fondamental des grammaires non contextuelles	109
8.3	Ambiguïté dans les grammaires	111
8.4	Arbre d'analyse	112
8.5	Les trucs que je n'ai pas le droit de vous raconter parce que c'est écrit hors programme, mais je le fais quand même	112
8.5.1	Décider si $u \in L(\mathcal{G})$	112
8.5.2	Automates à pile	113
8.5.3	Lemme de l'étoile pour les langages algébriques (Exercice 8.5)	113
	Exercices	113
9	Apprentissage et I.A.	117
9.0	Dans la vie, tout est un vecteur de \mathbb{R}^n	117
9.1	Apprentissage supervisé	117
9.1.1	Algorithme des k plus proches voisins (k -NN)	117
9.1.2	Matrice de confusion	120
9.1.3	ID3 et arbres de décision	120
9.2	Apprentissage non supervisé	121
9.2.1	CHA : Classification hiérarchique et ascendante	121
9.2.2	Algorithme des k -moyennes	122
9.3	Algorithme A*	122
	Table des algorithmes	125
	Références	127
II	Travaux pratiques	129
0	Tas binomiaux	131
0.1	Préparation de l'arborescence de travail	131
0.2	Arbres binomiaux	132
0.3	Arbres en tas minimum	132
0.4	Tas binomiaux	132
0.4.1	Définition et opérations élémentaires	132
0.4.2	Réunion de deux tas binomiaux	133
0.4.3	Insertion et suppression	134
0.4.4	Modification de la clé	134
0.4.5	Complexités	134
0.5	Applications	134
0.5.1	Révisions	134
0.5.2	S'asseoir loin	134
0.6	Point de vue du programmeur C	135

1	Jeu de Hex	137
1.1	Règles du jeu	137
1.2	Petits résultats théoriques	137
1.3	Programmer le jeu de Hex	137
1.4	Élagage $\alpha\beta$	138
1.5	Variante du jeu	138
2	Manipulations autour des expressions régulières et des langages locaux	139
2.1	Langages locaux	139
2.1.1	Reconnaissance d'un langage local	139
2.1.2	Opérations sur les langages locaux	140
2.2	Expressions régulières à la Kleene	140
2.3	Expressions régulières étendues en C	142
2.3.1	Formats de dates	143
2.3.2	Bannissez-les!	143
2.3.3	Ostracisme industrialisé	143
3	Union-find	145
3.1	Définition et implémentation	145
3.2	Étude de la complexité	146
3.3	Fabrication de labyrinthes	146
3.4	Faire la course	147
4	Couplage maximum biparti	149
4.1	Introduction	149
4.2	Couplages	149
4.3	Sommets libres	150
4.4	Chemins améliorants	150
4.5	Couplage maximum	150
4.6	Couplage parfait de poids maximum	151
4.7	Un peu plus loin	151
5	Petits jeux avec les processus légers	153
6	Résolution de SAT	157
6.1	Structures de données	157
6.2	Simplification élémentaire	157
6.3	Règle de propagation unitaire	157
6.4	Règle du littéral infructueux	158
6.5	Résoudre SAT	159
6.6	Transition de phase	159
7	Commutation d'interrupteurs	161
7.1	Parties d'un ensemble	161
7.1.1	Énumération des parties par incrément	162
7.2	Énumération des parties par un code de Gray	162
7.3	Sac à dos	163
7.3.1	Utilisation du calcul des parties d'un ensemble	163
7.3.2	Une stratégie alternative d'énumération	163
7.3.3	Un calcul de complexité	164
8	Approximation commerciale	165
8.1	Introduction	165
8.2	Représentation des graphes et des tours	165
8.3	Heuristique du plus proche voisin	165
8.4	Amélioration itérative	166
8.5	Évaluation et séparation	166

9 Automates finis	169
9.1 Représentation d'une machine à états	169
9.2 Automate déterministe	169
9.3 Lecture d'un automate	170
9.4 Langages locaux	170
9.5 Automates non déterministes	170
10 Algorithme de McNaughton et Yamada	171
10.1 Description de l'algorithme	171
10.2 Simplification d'expressions régulières	171
10.3 Programmation de l'algorithme de McNaughton et Yamada	172
10.4 Pour aller plus loin	172
10.5 Algorithme de Brzozowski et McCluskey	173
11 Chemin hamiltonien probabiliste	175
11.1 De l'aléatoire	175
11.2 Générer un graphe aléatoire, modèle d'Erds et Rényi	175
11.3 Un premier algorithme de chemin hamiltonien	176
11.4 Intermède mathématique	177
11.5 Preuve de validité	177
12 Assistant de preuve	179
12.1 Renommer les variables liées	179
12.2 Vérificateur d'arbre de preuve	180
12.3 Résoudre l'unification	180
13 Analyse d'un mini langage	181
13.1 Mini langage des expressions arithmétiques	181
13.2 Analyse lexicale	181
13.3 Analyse syntaxique	182
13.4 Commentaires	182
13.5 Liaison de variables	182
13.6 Pression de registres	183
14 Être à la mode	185
14.1 Code déjà tout prêt	185
14.2 Base du MNIST	185
14.3 File de priorité	186
14.4 k plus proches voisins naïf	186
14.5 Arbres dimensionnels	186
III Apartés culturels (HP)	187
A Matroïdes	189
A.0 Définition et exemples	189
A.1 Algorithme Glouton sur un matroïde	190
B Machines de Turing	193
B.0 Définition	193
C CCS et le PI calcul	195
D Dédution naturelle sur les types	197
D.0 Dédution des types en OCaml	197
D.1 Calcul avec le tiers exclu et continuations	199
D.1.1 Allégorie du diable	199
D.1.2 Continuations	199

E Révisions	201
E.0 Branch and bound (évaluation et séparation)	201
E.1 SQL, Bases de données	203
E.1.1 Dans une base de données, il y a des tables	203
E.1.2 Rappels sur les requêtes	203
E.1.3 Correction de l'exercice KeyVentou	203

Première partie

Cours

Chapitre 0

Jeux

Sommaire

0.0	Banalités	11
0.1	Définitions et vocabulaire	11
0.2	Attracteurs	13
0.3	Algorithme Min-Max	15
0.4	Algorithme d'élagage $\alpha\beta$	15

On racontera beaucoup d'histoires concernant certains personnages nommés Alice et Bob. Ce sont des noms fictifs et toute vraisemblance avec des faits réels ne saurait qu'être fortuite.

0.0 Banalités

Il s'agit d'étudier des interactions qu'ont des agents entre eux, par exemples, certains gagnent et d'autres perdent.

Dans ce chapitre, on étudie des jeux :

- à deux joueurs,
- à information complète,
- à somme nulle (les sommes des gains des deux joueurs est nulle),
- d'accessibilité.

Dans ces jeux, on trouve : le jeu de Hex, le jeu de go, les échecs, le morpion, le Reversi, le Puissance 4.

On ne trouve pas la belote (un joueur ne connaît pas l'état complet du jeu, les cartes des autres joueurs sont cachées), ni le tarot. On ne trouve pas non plus la roulette (pas de hasard).

0.1 Définitions et vocabulaire

Définition 1 (Jeu). Soit $V := V_A \sqcup V_B$ un ensemble partitionné. On appelle *jeu* un graphe $G := (V, E)$ et

- V_A est l'ensemble des sommets contrôlés par le premier joueur (Alice),
- V_B ceux du second (Bob).

Remarques. (i) Ainsi, les sommets représentent des *états* du jeu et les arêtes les *coups* effectués par les joueurs pour arriver dans d'autres positions.

(ii) On note au passage que le graphe obtenu n'est pas un arbre. Il n'est pas non plus biparti (même si on a deux ensembles de sommets disjoints) car on n'impose pas à priori qu'un coup passe d'un état contrôlé par un joueur à un état contrôlé par l'autre joueur.

(iii) Aucune contrainte, dans la définition, n'a été faite sur le caractère alterné du jeu. Autrement dit, on n'a aucune condition du type

$$\forall (e_1, e_2) \in E, \quad e_1 \in V_A \implies e_2 \in V_B.$$

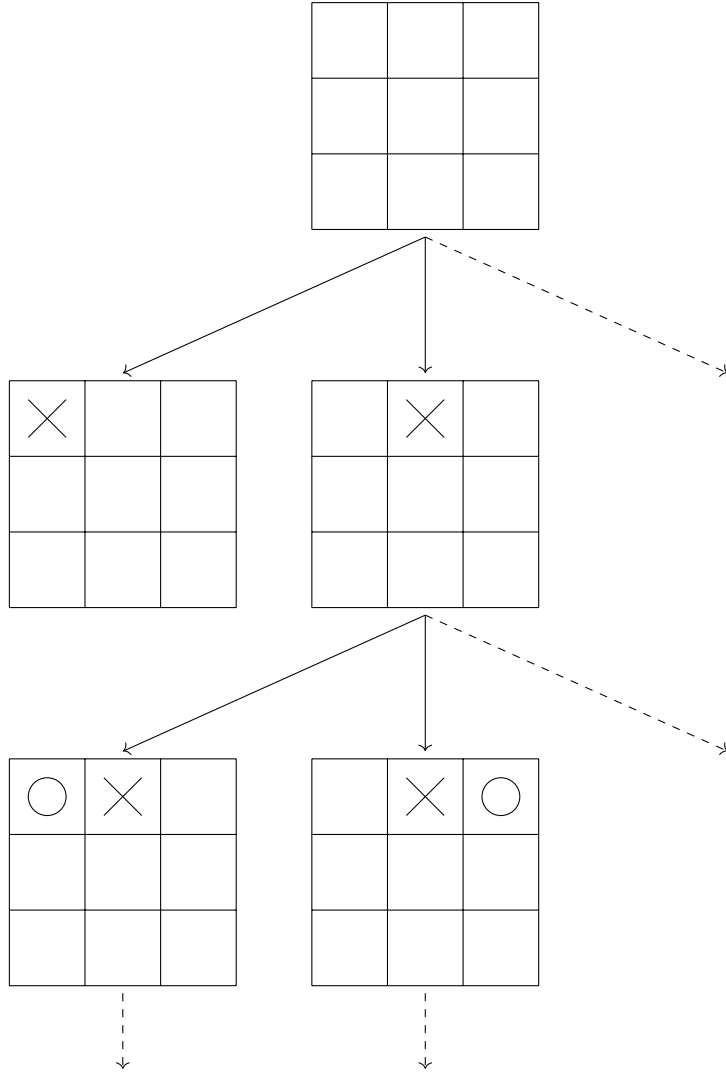


FIGURE 1 – Graphe du jeu du morpion

Théorème 2. *Tout jeu non alterné peut être transformé en un jeu alterné en rajoutant des états. Le graphe devient alors biparti.*

Démonstration. Pour chaque coup $(e_1, e_2) \in E$ tel que les états e_1 et e_2 sont contrôlés par le même joueur, on supprime ce coup, on ajoute un nouvel état f_{e_1, e_2} contrôlé par l'autre joueur, et on ajoute deux coups (e_1, f_{e_1, e_2}) et (f_{e_1, e_2}, e_2) , qui sont bien des coups alternant entre deux joueurs différents. Les nouveaux coups créés n'apparaissant que dans ces deux arêtes créées, on assure que le second joueur soit contraint à jouer le coup (f_{e_1, e_2}, e_2) , ce qui revient à lui faire passer son tour, ou encore à faire jouer le premier joueur deux fois de suite. ■

Ce théorème permet dès lors de faire l'hypothèse que tous les jeux que nous considérerons sont désormais alternants.

Définition 3 (Partie). Une partie est un chemin dans un jeu (potentiellement infini). Formellement, il s'agit d'un élément

$$(p_i) \in V^\omega \text{ où } V^\omega := \left(\bigcup_{k \in \mathbb{N}} V^k \right) \cup V^\mathbb{N} \text{ tel que } \forall i, (p_i, p_{i+1}) \in E$$

Définition 4 (Condition de gain). Une condition de gain pour un joueur est un sous-ensemble de V^ω

Définition 5 (Jeu d'accessibilité). Un jeu d'accessibilité est un jeu où l'on fixe un ensemble d'état $F \subseteq V$ gagnant pour un joueur et la condition de gain pour un joueur est «la partie passe par un sommet de F ».

- Remarques.* (i) On fixe F_A l'ensemble des états gagnants pour Alice (respectivement F_B ceux de Bob, remarquez que l'on a $F_A \cap F_B = \emptyset$). Une partie est gagnante pour Alice si et seulement si elle passe par un sommet de F_A avant tout sommet de F_B . On gardera ces notations dans la suite.
- (ii) Il peut exister des chemins où aucun ne gagne. Néanmoins, on peut rendre un jeu fini en remplaçant les cycles par des sommets spéciaux.

0.2 Attractions

Définition 6 (Attraction). Posons la suite de parties de V suivante :

$$A_0 := F_A, \quad \forall i \in \mathbb{N}, \quad A_{i+1} := A_i \cup \{e \in V_A : \exists (e, f) \in E, f \in A_i\} \\ \cup \{e \in V_B : \forall (e, f) \in E, f \in A_i \wedge \exists (e, f) \in E\}.$$

On définit alors l'*attraction* d'Alice comme

$$W_A := \bigcup_{i \in \mathbb{N}} A_i.$$

On définit de manière symétrique l'attraction de Bob.

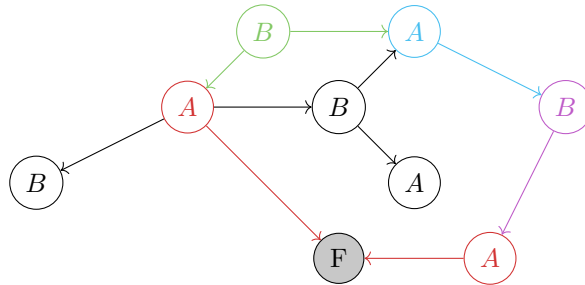
- Remarques.* (i) Pour comprendre intuitivement la définition précédente, l'attraction d'un joueur représente l'ensemble des états pour lesquels si ce joueur est dans son attracteur et joue de manière correcte, il gagnera. On construit récursivement cet ensemble : on part des états gagnants pour Alice, puis on ajoute les états contrôlés par Bob pour lesquels tous les successeurs sont déjà dans l'attraction (Bob n'a pas le choix) puis on ajoute les états contrôlés par Alice ayant au moins un successeur qui soit dans l'attraction (Alice choisit le meilleur coup), puis on recommence avec Bob.
- (ii) En outre, sans la toute dernière condition, il peut y avoir un puits contrôlé par Bob qui n'est pas gagnant pour Alice.
- (iii) La suite $(A_i)_{i \in \mathbb{N}}$ est, d'après sa définition, croissante pour l'inclusion. Pour tout $i \in \mathbb{N}$, A_i est fini et inclus dans V fini, de sorte que la suite $(A_i)_{i \in \mathbb{N}}$ est nécessairement stationnaire à partir d'un certain rang, c'est-à-dire qu'il existe un rang j tel que pour tout $i \geq j$, $A_i = W_A$.

Exemple. On considère le jeu de Nim : On dispose n bûchettes côte à côte, et au tour de chaque joueur, celui qui prend 1, 2 ou 3 bûchettes. Celui qui prend la dernière perd le jeu. L'attraction des deux joueurs et le même : c'est l'ensemble des états comportant k bûchettes où $k \not\equiv 1 \pmod{4}$ (le lecteur s'en convaincra aisément).

Définition 7 (Rang). Soit $v \in W_A$. On appelle rang de v l'entier

$$\text{rg}(v) := \min\{i \in \mathbb{N} : v \in A_i\}.$$

Exemple. On considère le jeu suivant où le noeud étiqueté F est une condition de victoire pour Alice :



On a représenté A_0 en gris, A_1 en rouge, A_2 en violet, A_3 en bleu, A_4 en vert et l'attraction d'Alice est ici l'ensemble des noeuds colorés (on a également coloré les arêtes qui explique l'appartenance d'un noeud).

Définition 8 (Stratégie). Une *stratégie* pour Alice est une fonction $\varphi : V_A \rightarrow V_B$ telle que

$$\forall v \in V_A, \quad (v, \varphi(v)) \in E.$$

Ainsi, on dit qu'une partie $(p_i)_{i \in \mathbb{N}}$ suit la stratégie φ si

$$\forall i \in \mathbb{N}, p_i \in V_A \implies p_{i+1} = \varphi(p_i).$$

φ est une *stratégie gagnante* pour Alice depuis l'état e si toute partie qui commence par e et qui suit φ est gagnante.

Théorème 9. *Avec l'attracteur, on peut construire une stratégie gagnante depuis tout état de celui-ci. On définit la stratégie φ pour tout $e \in V_A$ par :*

$$\varphi(e) := \begin{cases} f & \text{si } e \in W_A \text{ où } f \in W_A, (e, f) \in E \text{ et } \text{rg}(f) < \text{rg}(e) \text{ (existe par définition)} \\ - & \text{si } e \notin W_A \text{ (coup quelconque)} \end{cases}$$

Démonstration. Soit p une partie qui suit φ et qui démarre par un état $p_0 \in W_A$. On suppose que le jeu alterne.

Par récurrence sur $n \in \mathbb{N}$, on montre que $p_n \in W_A$.

- $p_0 \in W_A$ par hypothèse.
- Supposons le résultat acquis au rang n , on distingue les cas suivants :
 - si $p_n \in W_A$ est gagnant, la partie est gagnée.
 - si $p_n \in V_B$, alors par définition de W_A , tous les états f tels que $p_n, f \in E$ sont dans W_A , donc $p_{n+1} \in W_A$.
 - si $p_n \in V_A$, il n'est pas gagnant et par construction de φ , $\varphi(p_n) = p_{n+1} \in W_A$.

On extrait la sous-suite de (q_n) formée uniquement des coups d'Alice. Alors, $(\text{rg}(q_n))$ est une suite strictement décroissante d'entiers, donc elle est stationnaire. Notons n le rang à partir duquel elle est constante. Si $r := \text{rg}(q_n) > 0$, alors $q_n \in \{e \in V_A : \exists f \in A_{r+1}, (e, f) \in E\}$ car q_n est contrôlé par Alice. Donc $q_{n+1} = \varphi(q_n) \in V_B$. Ainsi, soit q_{n+1} est gagnant pour Alice, sinon il existe q_{n+2} tel que (q_{n+1}, q_{n+2}) est un coup. Mézamor, $\text{rg}(q_{n+2}) < \text{rg}(q_n)$, ce qui est absurde. On a alors nécessairement $\text{rg}(q_n) = 0$, d'où l'on déduit que q_n est gagnant. φ est donc une stratégie gagnante. ■

Algorithme 0.1 Calcul de l'attracteur

Entrée : Graphe $G = (V_A \sqcup V_B, E)$

Sortie : Tableau de booléens **win** qui à chaque sommet indique s'il est dans l'attracteur.

fonction TRAITER(s) :

Si non $\text{win}(s)$ **alors**

$\text{win}(s) \leftarrow \text{Vrai}$

Pour $p \in \text{pred}(s)$, **faire** :

$\text{nb_succ}(p) \leftarrow \text{nb_succ}(p) - 1$

Si $p \in V_A$ **ou** $\text{nb_succ}(p) = 0$ **alors**

 TRAITER(p)

Fin si

Fin pour

Fin si

Fin fonction

Pour $s \in V$, **faire** :

▷ Initialisation

$\text{win}(s) \leftarrow \text{Faux}$

$\text{nb_succ}(s) \leftarrow 0$

$\text{pred}(s) \leftarrow \emptyset$

Fin pour

Pour $(s, t) \in E$, **faire** :

$\text{nb_succ}(s) \leftarrow \text{nb_succ}(s) + 1$

$\text{pred}(t) \leftarrow \text{pred}(t) \cup \{s\}$

Fin pour

Pour $s \in F_A$, **faire** :

 TRAITER(s)

▷ Parcours depuis les états gagnants

Fin pour

0.3 Algorithme Min-Max

On cherche à marquer chaque noeud e du jeu d'une étiquette $\ell(e) \in \{-1, 0, 1\}$. Pour chaque $e \in E$ Premier étiquetage possible :

$$\ell(e) := \begin{cases} 1 & \text{si } e \in F_A \text{ (Noeud gagnant pour Alice)} \\ -1 & \text{si } e \in F_B \text{ (Noeud gagnant pour Bob)} \\ 0 & \text{sinon} \end{cases}$$

Deuxième étiquetage possible :

$$\ell(e) := \begin{cases} 1 & \text{si } e \in F_A \\ -1 & \text{si } e \in F_B \\ \max\{\ell(f) \mid (e, f) \in E\} & \text{si } e \in V_A \\ \min\{\ell(f) \mid (e, f) \in E\} & \text{si } e \in V_B \end{cases}$$

Cet algorithme marche bien avec les arbres, sur des DAG (encore mieux avec de la mémoïsation), mais ne converge pas en cas de cycle! Fichtre.

Pour limiter la profondeur de calcul de Min-Max, on se donne une heuristique : c'est une fonction $h : V \rightarrow \{-1, 0, 1\}$. On utilise h dans Min-Max en renvoyant $h(e)$ au lieu de calculer récursivement $\ell(e)$ si e est à profondeur plus grande qu'une profondeur fixée. On peut généraliser à une heuristique à valeurs dans $[-1, 1]$ (pour rappel, en informatique, $[-1, 1] = \llbracket -2^{31}, 2^{31} - 1 \rrbracket$).

0.4 Algorithme d'élagage $\alpha\beta$

On cherche à améliorer Min-Max. En effet, ce dernier effectue une exploration complète de l'arbre de recherche jusqu'à un niveau donné. L'élagage $\alpha\beta$ permet d'optimiser grandement l'algorithme minimax sans en modifier le résultat. Pour cela, il ne réalise qu'une exploration partielle de l'arbre. Lors de l'exploration, il n'est pas nécessaire d'examiner les sous-arbres qui conduisent à des configurations dont la valeur ne contribuera pas au calcul du gain à la racine de l'arbre. Dit autrement, l'élagage $\alpha\beta$ n'évalue pas des noeuds dont on peut penser, si la fonction d'évaluation est à peu près correcte, que leur qualité sera inférieure à celle d'un noeud déjà évalué.

Le principe est le suivant : on se donne à chaque instant deux bornes α et β . Un coup contrôlé par Alice de valeur inférieure à α est ignoré (il rapporte moins qu'un coup déjà exploré), Un coup contrôlé par Bob de valeur supérieure à β est ignoré (Bob ne jouera pas ce coup qui le désavantagerait trop). Pour un principe plus détaillé, se référer au TP 1 sur le jeu de Hex.

Algorithme 0.2 Élagage Alpha-Bêta

Entrée : Graphe $G = (V_A \sqcup V_B, E)$, Un état initiale $etat$, Profondeur de recherche maximale p

Sortie : Une estimations de qui va gagner depuis $etat$ dans $[-1, 1]$

```
fonction ALPHA-BETA-ALICE( $etat, p, \alpha, \beta$ ) :  
  Si  $etat \in F_A$  alors  
    renvoyer 1 ▷ Alice gagne  
  Sinon, si  $etat \in F_B$   
    renvoyer -1 ▷ Bob gagne  
  Sinon, si  $p = 0$   
    renvoyer  $h(etat)$  ▷ Fonction heuristique  
  Sinon  
     $maxscore \leftarrow -1$   
    Pour  $f \in V_B$  telle que  $(e, f) \in E$ , faire : ▷  $f$  est un coup licite  
       $valeur \leftarrow \text{ALPHA-BETA-BOB}(f, p - 1, \alpha, \beta)$   
       $\alpha \leftarrow \max\{\alpha, valeur\}$   
       $maxscore \leftarrow \max\{maxscore, valeur\}$   
      Si  $maxscore > \beta$  alors ▷ Élagage : Bob sait déjà faire mieux  
        renvoyer  $maxscore$   
      Fin si  
    Fin pour  
    renvoyer  $maxscore$   
  Fin si  
Fin fonction  
  
fonction ALPHA-BETA-BOB( $etat, p, \alpha, \beta$ ) :  
  Si  $etat \in F_A$  alors  
    renvoyer 1  
  Sinon, si  $etat \in F_B$   
    renvoyer -1  
  Sinon, si  $p = 0$   
    renvoyer  $h(etat)$   
  Sinon  
     $minscore \leftarrow +1$   
    Pour  $f \in V_A$  telle que  $(e, f) \in E$ , faire :  
       $valeur \leftarrow \text{ALPHA-BETA-ALICE}(f, p - 1, \alpha, \beta)$   
       $\beta \leftarrow \min\{\beta, valeur\}$   
       $minscore \leftarrow \min\{minscore, valeur\}$   
      Si  $minscore < \alpha$  alors ▷ Élagage : Alice sait déjà faire mieux  
        renvoyer  $minscore$   
      Fin si  
    Fin pour  
    renvoyer  $minscore$   
  Fin si  
Fin fonction  
  
Si  $etat \in V_A$  alors  
  renvoyer ALPHA-BETA-ALICE( $etat, p, -1, 1$ )  
Sinon, si  $etat \in V_B$   
  renvoyer ALPHA-BETA-BOB( $etat, p, -1, 1$ )  
Fin si
```

Chapitre 1

Langages

Sommaire

1.0	Définitions et vocabulaire	17
1.1	Opérations sur les langages	19
1.2	Question de régularité (ou rationalité)	19
1.2.1	Expressions régulières (ou rationnelles)	19
1.2.2	Langage d'une expression régulière	20
1.2.3	Expression linéaire	21
1.3	Expressions POSIX étendues	21
1.4	Langages locaux	21
1.4.1	Opérations sur les langages locaux (HP)	22
1.4.2	Lien avec les expressions régulières	22
	Exercices	23

1.0 Définitions et vocabulaire

Définition 1.1 (Monoïde). Un *monoïde* est un ensemble muni d'une multiplication associative et d'un neutre (le même à gauche et à droite).

Définition 1.2 (Alphabet). Un *alphabet* est un ensemble fini Σ . Ces éléments sont appelés *lettres*.

Définition 1.3. Σ^n sont les n -uplets d'éléments de Σ que l'on note $u_1 u_2 \cdots u_n$ au lieu de (u_1, u_2, \dots, u_n) .

Définition 1.4 (Concaténation). On définit la *concaténation*

$$\begin{aligned} \cdot : \quad & \Sigma^n \times \Sigma^p \rightarrow \Sigma^{n+p} \\ & (u_1 \cdots u_n, v_1 \cdots v_p) \mapsto u_1 \cdots u_n v_1 \cdots v_p \end{aligned}$$

Définition 1.5 (Mot). Un élément de Σ^n est un *mot*.

Définition 1.6 (Langage engendré). On définit le *langage engendré* par l'alphabet Σ :

$$\Sigma^* := \bigcup_{n \in \mathbb{N}} \Sigma^n$$

Cela donne une opération de concaténation interne

$$\cdot : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$



Remarque. $\Sigma^1 = \Sigma$ et $\Sigma^0 = \{\varepsilon\}$ où ε est le mot vide. Attention : $\Sigma^0 \neq \emptyset$!

Définition 1.7 (Longueur). Si $u \in \Sigma^*$, alors il existe un unique $n \in \mathbb{N}$ tel que $u \in \Sigma^n$. On note alors $|u|$ cet entier, appelé *longueur* de u . Si $u, v \in \Sigma^*$, on définit

$$u = v : \Longleftrightarrow \begin{cases} |u| = |v| \\ \forall i, \quad u_i = v_i \end{cases}$$

Proposition 1.8 (Neutre). Soit $u \in \Sigma^*$. On a $\varepsilon \cdot u = u = u \cdot \varepsilon$. Ainsi ε est neutre à gauche et à droite.

Proposition 1.9 (Associativité). Soit $u, v, w \in \Sigma^*$. On a

$$(u \cdot v) \cdot w = u \cdot (v \cdot w)$$

Démonstration. Notons $u_1 \cdots u_n$ les lettres de u , $v_1 \cdots v_p$ celles de v et $w_1 \cdots w_q$ celles de w . Alors d'après la définition de la concaténation \cdot , on obtient

$$(u \cdot v) \cdot w = (u_1 \cdots u_n v_1 \cdots v_p) \cdot w = u_1 \cdots u_n v_1 \cdots v_p w_1 \cdots w_q$$

De même avec $u \cdot (v \cdot w)$ où l'on trouve les mêmes lettres aux mêmes emplacements. ■

Remarque. Le triplet $(\Sigma^*, \cdot, \varepsilon)$ est un monoïde. C'est le monoïde libre engendré par Σ c'est-à-dire que l'on peut voir les lettres comparablement à une base pour un espace vectoriel. Faut-il préciser que c'est hors-programme ?

Définition 1.10 (Facteur, préfixe et suffixe). S'il existe quatre mots u, v, w et x tels que $u = v \cdot w \cdot x$, alors on dit que w est un *facteur* de u . Si $v = \varepsilon$, alors w est un *préfixe* de u . Si $x = \varepsilon$, alors w est un *suffixe* de u .

Proposition 1.11. (i) ε est préfixe et suffixe de tout mot.

(ii) Un mot est préfixe et suffixe de lui-même.

Exemple. Si $\Sigma = \{a, b\}$. $aaababbba$ est un mot de Σ^* . aaa en est un préfixe.

Définition 1.12 (Préfixe, suffixe et facteur propre ou strict). Un préfixe (resp. suffixe, resp. facteur) propre ou strict est un préfixe (resp. suffixe, resp. facteur) qui n'est pas le mot lui-même (donc de longueur strictement inférieure).

Définition 1.13. Soit $a \in \Sigma$. Si $u = u_1 \cdots u_n \in \Sigma^*$, on définit la *longueur en a*

$$|u|_a := \#\{j \in \mathbb{N} : u_j = a\}$$

Exemple. Considérons l'ensemble

$$L := \{a^n b^n\}_{n \in \mathbb{N}}$$

Les mots suivants appartiennent à L : ε , ab et $aaabbb$. On a également $|aaabbb|_a = 3 = |aaabbb|_b$.

On peut caractériser l'ensemble L par

$$u \in L \Longleftrightarrow \begin{cases} |u|_a = |u|_b \\ u \text{ ne contient pas le facteur } ba \end{cases}$$

Définition 1.14 (Langage). Un langage L sur Σ est un sous-ensemble de Σ^* .

Remarque. Le plus grand langage sur Σ est Σ^* et le plus petit est l'ensemble vide.

Exemples. Soit $\Sigma = \{a, b\}$ alors des langages sur Σ sont

- $L_0 = \emptyset$,
- $L_1 = \Sigma^*$,
- $L_2 = \{\varepsilon\}$,
- $L_3 = \{aba, ababaa\}$,
- $L_4 = \{a^n b^n\}_{n \in \mathbb{N}}$,
- $L_5 = \{a^n b^m\}_{(n,m) \in \mathbb{N}^2}$,
- $L_6 = \{a^{2^n}\}_{n \in \mathbb{N}}$,

- $L_7 = \{a^p\}_{p \in \mathbb{P}}$ (où \mathbb{P} est l'ensemble des nombres premiers),
 - $L_8 = \{u \in \Sigma^* : |u|_a = |u|_b\}$,
 - Langage de DYCK : les mots bien parenthésés (comme $abaabb$ qui correspond à $()(())$).
- Pour $\Sigma = \{a, b, c\}$, $L_{10} = \{a^n b^n c^n\}_{n \in \mathbb{N}}$ est un langage sur Σ .

Définition 1.15 (Palindrome). Si $u \in \Sigma^*$, on note $u = u_1 \cdots u_n$ et on définit le *retourné* de u le mot $\bar{u} = u_n \cdots u_1$. u est un *palindrome* si et seulement si $u = \bar{u}$. L'ensemble des palindromes sur un alphabet Σ est un langage.

1.1 Opérations sur les langages

Un langage est un ensemble, donc si L_1 et L_2 sont des langages sur un alphabet Σ alors $L_1 \cup L_2$ est un langage que l'on note $L_1 | L_2$ (ou $L_1 + L_2$). $L_1 \cap L_2$ est aussi un langage de même que $L_1 \setminus L_2$ et $\overline{L_1}$.

Définition 1.16 (Concaténation de deux langages). On définit la *concaténation* de L_1 et L_2 par

$$L_1 \cdot L_2 := \{u \cdot v \mid (u, v) \in L_1 \times L_2\}$$

Définition 1.17 (Étoile de KLEENE). Soit L un langage. On définit l'étoile de KLEENE de L comme

$$L^* := \bigcup_{n \in \mathbb{N}} L^n = L^0 \cup L^1 \cup L^2 \cup \dots$$

Exemple. Si $L = \{aba, ababaa\}$. Alors

$$L^* = \{\varepsilon, aba, ababaa, abaababaaaba, \dots\}$$

Remarque (Question du public). A-t-on $L^* \cdot L^* = L^*$ pour un langage L ? Par double inclusion :

\supseteq Soit $u \in L^*$. $\varepsilon \in L^*$ car $\varepsilon \in L^0$ donc $u \cdot \varepsilon \in L^* \cdot L^*$.

\subseteq Si $u \in L^* \cdot L^*$ alors il existe $v \in L^*$ et $w \in L^*$ tels que $u = v \cdot w$. $v \in L^*$ donc il existe $v_1, \dots, v_n \in L$ tel que $v = v_1 \cdot v_2 \cdots v_n$. De même : $w = w_1 \cdot w_2 \cdots w_p$. Finalement $u = v \cdot w \in L^{n+p} \subset L^*$.

Comme exercice, on pourra montrer que $(L^*)^* = L^*$.

Remarque. On a $\emptyset^* = \{\varepsilon\}$ et $\{\varepsilon\}^* = \{\varepsilon\}$.

Exemple. Dans l'exemple précédent : $L_6^* = \{a^n\}_{n \in \mathbb{N}}$.

Définition 1.18. Soit $u \in \Sigma^*$ et L un langage sur Σ . On définit $uL := \{u\} \cdot L$ et $u^{-1}L := \{v \in \Sigma^* : uv \in L\}$.

Exemple. Soit $L = \{a^{2n}\}_{n \in \mathbb{N}}$. On a $a^{-1}L = \{a^{2n+1}\}_{n \in \mathbb{N}}$. De plus, on a $(aa)^{-1}L = L$ et $(aaa)^{-1}L = a^{-1}L$.

Exemple. Soit $\Sigma = \{a, b\}$, $L = \{a^n b^m\}_{(n,m) \in \mathbb{N}^2}$ et $L' = \{u : |u|_a = |u|_b\}$. Calculer tous les $u^{-1}L$ et $u^{-1}L'$.

1.2 Question de régularité (ou rationalité)

1.2.1 Expressions régulières (ou rationnelles)

Définition 1.19 (Expression régulière). Soit un alphabet Σ . On définit le langage E des expressions régulières (*regex*) sur un alphabet $\Sigma \cup \{\emptyset, \varepsilon, (,), *, |, \cdot\}$ inductivement par :

$$\emptyset \in E, \quad \forall a \in \Sigma, \quad a \in E, \quad \varepsilon \in E,$$

et si $e_1, e_2 \in E$ alors

$$(e_1) \in E, \quad e_1 \cdot e_2 \in E, \quad e_1 * \in E, \quad e_1 | e_2 \in E.$$

Remarque. On écrit également la définition précédente comme

$$E = \{\emptyset\} \cup \{\varepsilon\} \cup \Sigma \cup (E \cdot E) \cup (E^*) \cup (E | E),$$

ce qui peut être pratique pour des démonstrations par induction structurelle.

Exemple. Pour $\Sigma = \{a, b\}$, \emptyset , $(\emptyset \cdot a) * | b$, $a \cdot a \cdot a$ sont des *regex*.

1.2.2 Langage d'une expression régulière

Définition 1.20 (Langage engendré par une expression régulière). On définit par une induction sur E une fonction :

$$\mathcal{L} : E \longrightarrow \mathcal{P}(\Sigma^*)$$

par :

$$\begin{aligned}\mathcal{L}(\emptyset) &:= \emptyset && \text{(ensemble vide)} \\ \mathcal{L}(\varepsilon) &:= \{\varepsilon\} && \text{(l'ensemble du neutre)} \\ \mathcal{L}(a) &:= \{a\} && \text{(où } a \in \Sigma) \\ \mathcal{L}((e)) &:= \mathcal{L}(e) \\ \mathcal{L}(e_1|e_2) &:= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\ \mathcal{L}(e^*) &:= \mathcal{L}(e)^* \\ \mathcal{L}(e_1 \cdot e_2) &:= \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)\end{aligned}$$

Exemple.

$$\mathcal{L}((\phi \cdot a)^* | b) = \{\varepsilon, b\}$$

Remarque. $\mathcal{L}(\emptyset \cdot _) = \emptyset$

Exemple.

$$\mathcal{L}(\mathbf{a^*b^*}) = \{a^n b^m\}_{(n,m) \in \mathbb{N}^2}$$

Définition 1.21 (Langage régulier). Un langage L est dit *régulier* s'il existe une regex e telle que $L = \mathcal{L}(e)$

Exemple. Sont des langages réguliers :

- $\{a^n b^m | n, m \in \mathbb{N}\} = \mathcal{L}(a^* b^*)$
- $\{a^n\}_{n \in \mathbb{N}^*} = \mathcal{L}(aa^*)$
- $\{u : |u|_a \bmod 3 = 0\} = \mathcal{L}((b^* ab^* ab^* ab^*)^* | b^*)$
- $\{u : |u|_a \bmod 3 = 0 \wedge |u|_b \bmod 3 = 0\}$, admis pour l'instant.

Définition 1.22 ($\text{Reg}(\Sigma)$). $\text{Reg}(\Sigma)$ est l'ensemble de tous les langages réguliers sur l'alphabet Σ

Proposition 1.23. La réunion, la concaténation, l'intersection de deux langages réguliers donne un langage régulier. L'étoile de KLEENE et le complémentaire d'un langage régulier est encore régulier. Ainsi $\text{Reg}(\Sigma)$ est stable par $\cup, \cap, \cdot, -, \overline{}$,

Démonstration. Soit K et L deux langages réguliers engendrés par les expressions régulières k et l respectivement.

- $\mathcal{L}(k^*) = \mathcal{L}(k)^* = K^*$.
- $\mathcal{L}(k|e) = \mathcal{L}(k) \cup \mathcal{L}(l) = K \cup L$.
- $\mathcal{L}(k \cdot l) = \mathcal{L}(k) \cdot \mathcal{L}(l) = K \cdot L$.

ce qui démontre le résultat voulu. On admet le résultat pour le complémentaire et pour l'intersection que l'on verra au chapitre 5. ■

Proposition 1.24. Tout langage fini L est régulier.

Démonstration. En effet, puisque L est fini, on peut écrire

$$\mathcal{L}\left(\bigcup_{u \in L} u\right) = \bigcup_{u \in L} \mathcal{L}(u) = \bigcup_{u \in L} \{u\} = L,$$

d'où le résultat. ■

Remarque. C'est même la plus petite classe de langages sur Σ qui contient le langage \emptyset et les singletons de ε et soit stable par ces opérations. (La plus grande étant $\mathcal{P}(\Sigma^*)$).

A-t-on $\text{Reg}(\Sigma) = \mathcal{P}(\Sigma^*)$? Non : en général, on a seulement $\text{Reg}(\Sigma) \subseteq \mathcal{P}(\Sigma^*)$. Par exemple, $\{a^n b^n\}_{n \in \mathbb{N}}$ et $\{a^p\}_{p \in \mathbb{P}}$ ne sont pas réguliers.



Si pour tout $n \in \mathbb{N}$, L_n est régulier, $\bigcup_{n \in \mathbb{N}} L_n$ ne l'est pas nécessairement ! Exemple avec $L_p := \{a^p\}$ si $p \in \mathbb{P}$ et $L_p := \emptyset$ sinon. Si L est régulier, $L' \subseteq L$ n'est pas non plus nécessairement : $\{a^p\}_{p \in \mathbb{P}} \subset \{a^n\}_{n \in \mathbb{N}}$. On a même que Σ^* régulier, or tout langage est dans $\mathcal{P}(\Sigma^*)$.

1.2.3 Expression linéaire

On raffine la notion d'expression régulière.

Définition 1.25 (Expression régulière linéaire). Une expression régulière sur un alphabet Σ est dite *linéaire* si toute lettre de Σ y apparaît au plus une fois.

Exemple. aa^* n'est pas linéaire. $(a * b)^*$ est linéaire. a^* est linéaire. $\varepsilon|a|aa^*$ n'est pas linéaire.

Étant donné une expression régulière quelconque, on cherche désormais à la transformer (quitte à changer l'alphabet de départ) en une expression régulière linéaire. Ce processus se nomme linéarisation.

Soit e une expression régulière sur Σ , on note N le nombre de lettres apparaissant dans e . On change l'alphabet Σ en $\Sigma' := \Sigma \times \llbracket 1, N \rrbracket$ et on numérote chaque lettre de e afin d'avoir un couple lettre numéro pour obtenir une expression régulière, notée $\text{lin}(e)$, sur Σ' qui est linéaire.

Cette opération peut, en quelque sorte, être renversé par un «oubli» :

$$U : \begin{array}{ccc} \Sigma' & \longrightarrow & \Sigma \\ (l, n) & \longmapsto & l \end{array}$$

qui s'étend en une fonction sur $\text{Reg}(\Sigma')$ et $(\Sigma')^*$. Par construction il vient alors $U(\text{lin}(e)) = e$.

Remarque. Le choix de la définition de Σ' est totalement arbitraire, on aurait pu prendre une partie finie quelconque de $\Sigma \times \mathbb{N}$, il faut seulement qu'elle puisse numéroté de manière univoque les lettres de e . On donne un exemple ci-dessous.

Exemple (Une linéarisation valide). Si $e = e_0 \dots e_{n-1}$ est une expression régulière, alors on transforme chaque e_i en :

- e_i si e_i est un symbole d'expression régulière
- ou (e_i, i) si $e_i \in \Sigma$.

Dans ce cas, $\Sigma' = \Sigma \times \llbracket 0, n-1 \rrbracket$, $\text{lin}(e)$ est une expression régulière sur Σ' .

1.3 Expressions POSIX étendues

Une regex POSIX-étendue est une regex avec $| * ()$, mais aussi :

- $.$: accepte n'importe quel caractère
- $+$: cherche un motif présent au moins une fois (comme $*$ mais l'élément doit être présent au minimum une fois
- $?$: cherche un motif présent au plus une fois
- $[\dots]$: accepte un des caractères dans le crochet
- $[^ \dots]$: pour un caractère qui n'est pas dans la liste
- $[0-9]$: raccourci pour écrire $[0123456789]$
- $^$: pour le début de ligne
- $\$$: pour la fin de ligne

Exemple. $[a-zA-Z_][a-zA-Z0-9_]*$ matche les noms de variables ou de fonctions en C.

Remarque. On peut échapper les caractères spéciaux en ajoutant un \backslash avant : ainsi $\backslash*$ matche la chaîne de caractères $"*"$.

1.4 Langages locaux

Définition 1.26. Un langage L est dit *local* lorsqu'il est décrit par un quadruplet (P, D, F, α)

- $P \subset \Sigma$, de premières lettres autorisées en début de mot,
- $D \subset \Sigma$, un ensemble de dernières lettres autorisées,
- $F \subset \Sigma^2$ un ensemble de facteurs de longueur 2 autorisés (c'est-à-dire que tous facteurs de longueur 2 d'un mot de L doit être dans F),
- α un booléen indiquant si L contient ε ,

Exemple.

1.4.1 Opérations sur les langages locaux (HP)

Proposition 1.27 (Stabilité de la classe des langages locaux). *Soient L, L_1, L_2 des langages locaux sur les alphabets $\Sigma, \Sigma_1, \Sigma_2$.*

- (i) *Stabilité par intersection : $L_1 \cap L_2$ est local*
- (ii) *Stabilité par étoile de KLEENE : L^* est local*

Démonstration. Il suffit d'exhiber les langages solutions :

- (i) Soit $L_1 := (P_1, D_1, F_1, \alpha_1)$ et $L_2 := (P_2, D_2, F_2, \alpha_2)$:
 $L_1 \cap L_2 = (P_1 \cap P_2, D_1 \cap D_2, F_1 \cap F_2, \alpha_1 \wedge \alpha_2)$ sur l'alphabet $\Sigma_1 \cap \Sigma_2$
- (ii) Soit $L := (P, D, F, \alpha) : L^* = (P, D, F \cup (D \times P), \top)$

■

Remarque. En revanche, les autres opérations usuelles sur les langages ne se transfèrent pas *a priori* sur les langages locaux, comme le montre les contre-exemples suivants pour l'union et la concaténation.

- Pour l'union, après une rapide analyse (cf. preuve de la proposition 1.28), on obtiendrait comme candidat pour $L_1 \cup L_2$ le langage local $(P_1 \cup P_2, D_1 \cup D_2, F_1 \cup F_2, \alpha_1 \vee \alpha_2)$. Le problème vient des facteurs : pour un mot dans $L_1 \cup L_2$, ses facteurs sont soit tous dans F_1 , soit tous dans F_2 , or avec ce candidat, il serait possible que les facteurs soient dans F_1 et F_2 . On obtient un contre-exemple avec les langages $L_1 = (\{a\}, \{b\}, \{aa, ab, bb\}, \perp) = \{a^n b^m\}_{(n,m) \in \mathbb{N}^{*2}}$, et $L_2 = (\{b\}, \{a\}, \{aa, ba, bb\}, \perp) = \{b^n a^m\}_{(n,m) \in \mathbb{N}^{*2}}$.
- Pour la concaténation, l'analyse est légèrement plus complexe, mais on obtient le même contre-exemple que ci-dessus.

On explore, en revanche, dans la suite quelques conditions qui permettent d'en obtenir par ces mêmes opérations.

Proposition 1.28 (Condition suffisante sur l'union de langages locaux). *Soit L_1 et L_2 deux langages locaux sur deux alphabets Σ_1 et Σ_2 . Pour que $L_1 \cup L_2$ soit local, il suffit que $\Sigma_1 \cap \Sigma_2 = \emptyset$*

Démonstration. On procède par analyse synthèse.

- *Analyse* : Supposons que $L := L_1 \cup L_2$ soit local. Ainsi $P_1 \subset P$ et $P_2 \subset P$ de sorte que $P_1 \sqcup P_2 \subset P$. On fait de même pour D et F . On observe finalement que $\alpha = \alpha_1 \vee \alpha_2$ nécessairement.
- *Synthèse* : on vérifie réciproquement que si l'on pose
 - $P := P_1 \sqcup P_2$,
 - $D := D_1 \sqcup D_2$,
 - $F := F_1 \sqcup F_2$,
 - $\alpha := \alpha_1 \vee \alpha_2$,
 on fournit à $L := L_1 \sqcup L_2$ une structure de langage local

■



Remarque. Attention cependant, les expressions obtenues ci-dessus sont trompeuses.

Proposition 1.29 (Condition suffisante sur la concaténation de langages locaux). *Soit L_1 et L_2 deux langages locaux sur deux alphabets Σ_1 et Σ_2 . Pour que $L_1 \cdot L_2$ soit local, il suffit que $\Sigma_1 \cap \Sigma_2 = \emptyset$*

Démonstration. On procède par analyse synthèse, la preuve sera similaire à celle précédente.

- *Analyse* :
- *Synthèse* : On vérifie aisément qu'avec les définitions précédentes, on fournit à $L := L_1 \cdot L_2$ une structure de langage local.

■

1.4.2 Lien avec les expressions régulières

Proposition 1.30. *Un langage local est régulier.*

Démonstration. Soit $L := (P, D, F, \alpha)$ un langage local. Observons alors que

$$L \setminus \{\varepsilon\} = (D\Sigma^* \cap \Sigma^*F) \setminus \Sigma^*F\Sigma^*.$$

$L \setminus \{\varepsilon\}$ est donc régulier par opérations sur les langages réguliers. Il suffit alors, en fonction de la valeur de α , de rajouter ou non ε ce qui est sans effet sur la régularité de l'expression précédente, et permettant d'obtenir L .

■

On s'intéresse dans ce qui suit au lien entre les expressions régulières linéaires et les langages locaux

Proposition 1.31. *Une expression régulière linéaire engendre un langage local*

Démonstration. Par induction sur la structure de l'expression régulière. Soit \mathcal{P} la propriété définie sur $e \in \text{Reg}(\Sigma)$ linéaire, par " $\mathcal{L}(e)$ est local".

- Si $e = \emptyset$ alors $\mathcal{L}(e) = \emptyset$ est le langage local décrit par $(\emptyset, \emptyset, \emptyset, \perp)$.
- Si $e = \varepsilon$ alors $\mathcal{L}(e) = \{\varepsilon\}$ c'est le langage local engendré par $(\emptyset, \emptyset, \emptyset, \top)$.
- Si $e \in \Sigma$ alors $\mathcal{L}(e) = \{e\}$. C'est le langage local engendré par $(\{e\}, \{e\}, \emptyset, \perp)$.
- Si $e = e_1 e_2$. Par hypothèse d'induction, $\mathcal{L}(e_1), \mathcal{L}(e_2)$ sont locaux, et puisque e est linéaire, on peut définir e_1 et e_2 sur deux alphabets disjoints, de sorte que $\mathcal{L}(e_1), \mathcal{L}(e_2)$ puissent être considérés sur deux alphabets disjoints. En ces conditions, la proposition 1.28 garantit que $\mathcal{L}(e) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$ est local.
- De même pour $e_1 \cdot e_2$ et e^* .

Donc d'après le principe d'induction sur les expressions régulières, si e est linéaire, alors $\mathcal{L}(e)$ est local ce qu'on voulait. ■

Exercices

Exercice 1.1. Soient L, M et N trois langages sur un alphabet Σ . Comparer (inclusion, égalité) les langages suivants :

- $L \cdot (M \cap N)$ et $L \cdot M \cap L \cdot N$;
- $L \cdot (M \cup N)$ et $L \cdot M \cup L \cdot N$;
- $(\overline{L})^*$ et $\overline{L^*}$;
- L^* et $(L^*)^*$;
- $(L \cup M)^*$ et $(L^* M^*)^*$;
- $L^* \mid M^*$ et $(L \mid M)^*$;
- $(L^2)^*$ et $(L^*)^2$;
- $(\Sigma^2)^*$ et $(\Sigma^*)^2$.

Exercice 1.2 (Mots bien parenthésés). On se place sur l'alphabet $\Sigma = \{a, b\}$. Le langage L_P des mots bien parenthésés est le langage défini inductivement par la relation :

$$L_P = \{\varepsilon\} \cup aL_P b \cup L_P \cdot L_P$$

1. Démontrer soigneusement que $aabaabbb$ est bien parenthésé et que ba et abb ne le sont pas.
2. Démontrer que si un mot est bien parenthésé alors il existe une bijection qui associe chaque a à un b qui lui est postérieur.
3. Montrer que pour tout $w \in L_P$, on a $|w|_a = |w|_b$, et que si $w \neq \varepsilon$, alors il commence par un a et termine par un b .
4. Montrer que lorsque $w \in L_P$, alors pour tout i tel que $w_i = a$, il existe $j > i$ tel que $w_i \cdots w_j \in L_P$, où $w_i \cdots w_j$ est le sous-mot de w constitué de ses lettres d'indices compris entre i et j . A-t-on unicité de j ?
5. Montrer soigneusement que $w \in L_P$ si et seulement si $|w|_a = |w|_b$ et, pour tout préfixe u de w , $|u|_a \geq |u|_b$.

Exercice 1.3 (Lemme d'Arden). On fixe un alphabet Σ . Si A et B sont deux langages sur cet alphabet, on notera $A \cdot B$ l'ensemble défini par :

$$A \cdot B = \{vw \mid v \in A, w \in B\}$$

Le lemme d'Arden permet de résoudre des équations de la forme $X = (A \cdot X) \mid B$, où A, B et X sont des langages.

1. Rappeler la définition du langage A^* , où A est un langage.

2. Soit I un ensemble non vide et, pour chaque $i \in I$, soit A_i un langage sur Σ . Soit B un langage sur Σ . Montrer les propositions suivantes :

$$\left(\bigcup_{i \in I} A_i\right) \cdot B = \bigcup_{i \in I} (A_i \cdot B) \qquad B \cdot \left(\bigcup_{i \in I} A_i\right) = \bigcup_{i \in I} (B \cdot A_i)$$

On désigne par (E) l'équation $X = (A \cdot X) \mid B$, où $A \subset \Sigma^*$ et $B \subset \Sigma^*$.

1. Démontrer que $A^* \cdot B$ est une solution de (E) .
2. Démontrer que, pour l'inclusion, $A^* \cdot B$ est la plus petite solution de (E) .
3. Donner un exemple de valeurs de A et B pour lesquelles l'équation (E) admet plusieurs solutions.
4. Démontrer que si $\varepsilon \notin A$, alors $A^* \cdot B$ est l'unique solution de (E) .

Exercice 1.4 (Bords). On appelle *bord* d'un mot u sur un alphabet Σ tout mot v non vide qui est à la fois préfixe et suffixe de u . Prouver que si Σ contient au moins deux lettres, alors pour tout mot u non vide sur Σ , il existe un mot v sur Σ tel que uv n'a pas de bord.

Exercice 1.5 (Résiduels). Soit L un langage sur un alphabet Σ et soit u un mot sur A . On appelle *résiduel à gauche de L par rapport à u* , et on note $u^{-1}L$, l'ensemble :

$$u^{-1}L = \{v \in \Sigma^* \mid uv \in L\}$$

1. On se place sur $\Sigma = \{a, b\}$. Calculer le résiduel à gauche des langages suivants par rapport à tout mot : $L_1 = \{a\}^* \{b\}^*$ puis $L_2 = \{a^n b^n \mid n \in \mathbb{N}^*\}$.
2. Si x est une lettre de Σ , que valent $x^{-1}(L \cup L')$, $x^{-1}(LL')$ et $x^{-1}L^*$?
3. On définit la relation \sim_L , appelée congruence de Nérède, sur les mots de Σ^* par : $u \sim_L v$ si et seulement si $u^{-1}L = v^{-1}L$. Montrer que \sim_L est une relation d'équivalence. Montrer quelle vérifie, pour tous mots u, v et $v, u \sim_L v \implies uv \sim_L vw$.
4. Sur $\Sigma = \{a, b\}$, on pose L l'ensemble des mots ayant un nombre de a multiple de 3. Dans chacun des cas suivants, déterminer s'ils sont équivalents par \sim_L :
 - (a) b et ab ;
 - (b) aba et bab ;
 - (c) $abbaba$ et aaa .

Exercice 1.6 (Lemme de Levy). Soient x, y, u et v sont quatre mots sur un alphabet Σ qui vérifient $xy = uv$. Montrer qu'il existe un mot $z \in \Sigma^*$ tel que $(u = xz \text{ et } y = zv)$ ou $(x = uz \text{ et } v = zy)$.

Exercice 1.7 (Commutant d'un mot). 1. Soit x et y deux mots tels que $xy = yx$. Montrer qu'il existe deux entiers naturels $p, q \in \mathbb{N}$ et $w \in \Sigma^*$ tels que $x = w^p$ et $y = w^q$.

2. Un mot est dit *primitif* lorsqu'il n'est la puissance d'aucun autre mot que lui-même. Montrer que tout mot non vide s'écrit de façon unique u^k où u est primitif et $k \in \mathbb{N}$. (Pour ε , on perd l'unicité.)

3. On fixe $w \in \Sigma^*$. Préciser la nature de l'ensemble des mots qui commutent avec w .

Exercice 1.8 (Mots de Fibonacci). On définit sur l'alphabet $\{a, b\}$ les mots de Fibonacci par $f_0 = \varepsilon$, $f_1 = a$, $f_2 = b$ puis par la relation de récurrence $f_{n+2} = f_{n+1}f_n$.

1. Déterminer la longueur de f_n en fonction de n .
2. Déterminer le nombre de a et de b de f_n .
3. Le mot f_n contient-il plusieurs a consécutifs ? Plusieurs b consécutifs ?
4. Pour plus tard : le langage formé des mots de Fibonacci est-il régulier ?

Exercice 1.9. On appelle *code* sur un alphabet Σ tout langage X sur Σ qui vérifie la propriété suivante : si x_1, \dots, x_p et y_1, \dots, y_q sont des mots de X tels que $x_1x_2 \dots x_p = y_1y_2 \dots y_q$ alors $p = q$ et pour tout i , $x_i = y_i$. Autrement dit, tout élément de X^* se factorise de manière unique sur X .

1. Les langages suivants sont-ils des codes ? $X_1 = \{ab, baa, abba, aabaa\}$; $X_2 = \{b, ab, baa, abaa, aaaa\}$; $X_3 = \{aa, ab, aab, bba\}$; $X_4 = \{a, ba, bba, baab\}$.

2. Soit u un mot de Σ^* . Montrer que $\{u\}$ est un code si et seulement si $u \neq \varepsilon$.
3. Soient u et v deux mots distincts de Σ^* . Montrer que la partie $\{u, v\}$ est un code si et seulement si u et v ne commutent pas.
4. Soit X une partie de Σ^* ne contenant pas ε et telle qu'aucun mot de X ne soit préfixe propre d'un autre mot de X . Montrer que X est un code. (Un tel code est appelé *code préfixe*.)

Exercice 1.10. Soit L un langage sur un alphabet Σ . On désigne par $FG(L)$ l'ensemble des facteurs gauches (préfixes) de L .

1. Trouver l'ensemble des facteurs gauches de chacun des langages suivants sur l'alphabet $\Sigma = \{a, b\}$:
 $L_1 = \{a^n b^n \mid n \geq 0\}$, $L_2 = \{a^n b^m \mid 0 \leq n \leq m\}$, $L_3 = \{a^n b^m \mid 0 \leq m \leq n\}$, $L_4 = \{u \in \Sigma^* \mid |u|_a = |u|_b\}$.
2. On appelle *centre de L* et on note $c(L)$ l'ensemble des mots u de Σ^* tels que $\text{card}(u\Sigma^* \cap L)$ soit infini. Calculer $c(L)$ pour les quatre langages ci-dessus.
3. Que valent $c(L \cup L')$, $c(LL')$, $c(LL^*)$ en fonction de $c(L)$ et $c(L')$?

Chapitre 2

Graphes

Sommaire

2.0	Révisions	27
2.1	Graphes bipartis	29
2.1.1	Couplage dans un graphe biparti	29
2.1.2	Lien aux flots dans les graphes (HP)	32
2.2	Graphes orientés acycliques (DAG)	33
2.2.1	Tri topologique	34
2.2.2	Composantes fortement connexes	35
2.2.3	2-SAT	38
2.3	Arbres couvrants	39
2.4	Distances dans les graphes	41
2.4.1	Définitions	41
2.4.2	DIJKSTRA	42
2.4.3	FLOYD-WARSHALL	43
2.4.4	Retour sur la programmation dynamique	45

2.0 Révisions

Définition 2.1 (Graphe). Un *graphe* est un couple $G := (V, E)$ où V est l'ensemble des sommets, E l'ensemble des arêtes.

Dans tout ce chapitre désormais, $G := (V, E)$ désigne un graphe.

Définition 2.2 (Graphe orienté). On dit que G est *orienté* si $E \subset V^2$.

Définition 2.3 (Graphe non orienté). On dit que G est *non orienté* si $E \subset \mathcal{P}_2(V)$.

Définition 2.4 (Degré). Soit $v \in V$. Si G est non orienté, on définit le *degré* de v par :

$$d^o(v) := \#\{e \in E : v \in e\}$$

Si G est orienté, on définit respectivement les *degré entrant* et *degré sortant* du sommet v par :

$$d^+(v) := \#\{(x, y) \in E : y = v\} \quad \text{et} \quad d^-(v) := \#\{(x, y) \in E : x = v\}$$

Proposition 2.5 (Poignées de main). Si G n'est pas orienté, alors

$$\sum_{v \in V} d^o(v) = 2\#E$$

Démonstration. Chaque arrête est comptée deux fois lorsque l'on compte le degré de deux noeuds voisins. On peut le formaliser de la manière suivante :

$$\sum_{v \in V} d^o(v) = \sum_{v \in V} \sum_{e \in E} \mathbb{1}_e(v).$$

les sommes étant finies, le théorème de Fubini s'applique et donne :

$$\sum_{v \in V} \sum_{e \in E} \mathbb{1}_e(v) = \sum_{e \in E} \sum_{v \in V} \mathbb{1}_e(v) = \sum_{e \in E} 2 = 2\#E,$$

où l'on a noté $\mathbb{1}$ la fonction caractéristique d'un ensemble en indice. ■

Définition 2.6 (Graphe k -régulier). On dit que G est k -régulier si $\forall v \in V, d^o(v) = k$

Exemple. Quelques exemples de graphes k -réguliers :

- K_4 est régulier
- $K_{3,3}$ aussi



FIGURE 2.1 – Exemples de graphes réguliers : $K_{3,3}$ et K_4

Proposition 2.7. Si G (non vide) est 2-régulier, alors il contient au moins un cycle.

Démonstration. On va montrer que le graphe possède un cycle élémentaire. Si $v_0 \in V$, il existe un voisin v_1 de v_0 . Par récurrence, si $(v_0, \dots, v_n, v_{n+1})$ est construit, G est 2 régulier donc il existe un sommet v_{n+2} distinct de v_n voisin de v_{n+1} . Ceci construit un chemin (infini). Or, le graphe est fini. On considère le premier rang j où on repasse par un sommet v_i déjà vu. Dans le sous chemin $(v_i, v_{i+1}, \dots, v_j)$, (avec $v_i = v_j$), les sommets v_i, \dots, v_{j-1} sont deux à deux distincts. Par construction, ce chemin n'est pas réduit à une seule arrête. Donc, c'est un cycle élémentaire. ■

Définition 2.8 (Arbre). On dit que G est un arbre si il est connexe et acyclique.

Lemme 2.9. Soit $m := \#E$. Pour $k \in \llbracket 0, m \rrbracket$, on définit $G_k := (V, \{a_0, a_1, \dots, a_{k-1}\})$, obtenu en ne conservant que les k premières arêtes de G . Alors dans le passage de G_k à G_{k+1} , ajouter a_k diminue le nombre de composantes connexes de 1 si, et seulement si l'ajout de a_k ne crée pas de cycle, et ne fait rien sinon.

Démonstration. Soit $k \in \llbracket 0, m-1 \rrbracket$. Pour tout graphe G , on note $\mathcal{C}(G)$ le nombre de composantes connexes de G .

- Si l'ajout de a_k ajoute un cycle, c'est que les extrémités de a_k sont dans la même composante connexe dans G_k . Donc a_k ne change pas le nombre de composantes connexes : $\mathcal{C}(G_{k+1}) = \mathcal{C}(G_k)$
- Sinon, c'est que les extrémités de a_k sont dans deux composantes connexes différentes de G_k . Alors l'ajout de a_k fusionne ces deux composantes en une seule : $\mathcal{C}(G_{k+1}) = \mathcal{C}(G_k) - 1$

■

Proposition 2.10. On dispose des caractérisations suivantes sur le nombre d'arêtes par rapport au nombre de sommets :

- (i) Si $\#E > \#V - 1$, alors G est cyclique.
- (ii) Si $\#E < \#V - 1$, alors G n'est pas connexe.

Démonstration. On utilise le lemme précédent (2.9), duquel on reprend les notations :

- (i) Par contraposée, supposons G est acyclique. Alors

$$\begin{aligned} \#V - \#E &= \mathcal{C}(G_0) - m \\ &= \mathcal{C}(G_m) \quad \text{car } G \text{ acyclique (Récurrence sur le lemme)} \\ &= \mathcal{C}(G) \geq 1 \end{aligned}$$

Donc $\#E \leq \#V - 1$. Ainsi si $\#E > \#V - 1$, G est nécessairement cyclique.

- (ii) Supposons $\#E < \#V - 1$.

$$\mathcal{C}(G) = \mathcal{C}(G_m) \geq \mathcal{C}(G_0) - m = \#V - \#E \geq 2$$

En effet, le lemme assure qu'en passant de G_0 à G_m , on a retiré au plus m fois 1 composante connexe (moins si des cycles ont été créés). Donc G possède donc au moins deux composantes connexes, donc ne peut être connexe.

■

Théorème 2.11. *Les assertions suivantes sont équivalentes :*

- (i) G est un arbre.
- (ii) G est connexe et $\#E = \#V - 1$.
- (iii) G est acyclique et $\#E = \#V - 1$.

Démonstration. On s'appuie encore sur le lemme 2.9.

(i) \Rightarrow (ii) \wedge (iii). Supposons que G est un arbre. D'après 2.10, d'une part G est acyclique donc $\#E \leq \#V - 1$ et d'autre part il est connexe donc $\#E \geq \#V - 1$. Finalement, $\#E = \#V - 1$.

(ii) \Rightarrow (i). On suppose G connexe et $\#E = \#V - 1$. Montrons alors qu'il est acyclique. En reprenant les notations du lemme, on sait que $\mathcal{C}(G_m) = \mathcal{C}(G) = 1$ car G est connexe. Or on a $\mathcal{C}(G_0) = \#V = m + 1$. On a donc $\mathcal{C}(G_0) - \mathcal{C}(G_m) = m$, ce qui veut dire qu'en ajoutant m arêtes, on a enlevé m composantes connexes, c'est-à-dire que toute arête ajoutée a diminué de 1 le nombre de composantes connexes. D'après le lemme, aucune arête n'a donc ajouté de cycle. D'où l'on tire que G est acyclique et est donc un arbre.

(iii) \Rightarrow (i). On suppose G acyclique et $\#E = \#V - 1$. Montrons alors qu'il est connexe.

$$\begin{aligned} \mathcal{C}(G) &= \mathcal{C}(G_m) = \mathcal{C}(G_0) - m && \text{car } G \text{ est acyclique} \\ &= \#V - (\#V - 1) = 1 \end{aligned}$$

D'où G est connexe, c'est donc un arbre.

■

Algorithme 2.1 Reconnaissance d'un arbre

Entrée : Graphe $G = (V, E)$

Sortie : Booléen `est_arbre` qui indique par sa valeur si G est un arbre

2.1 Graphes bipartis

Définition 2.12 (Graphe biparti). On dit que G est *biparti* s'il existe V_1 et V_2 tels que $V = V_1 \sqcup V_2$ et pour toute arête de E entre deux sommets s et t ,

$$s \in V_1 \iff t \in V_2.$$

Exemple. Voici quelques graphes bipartis célèbres :

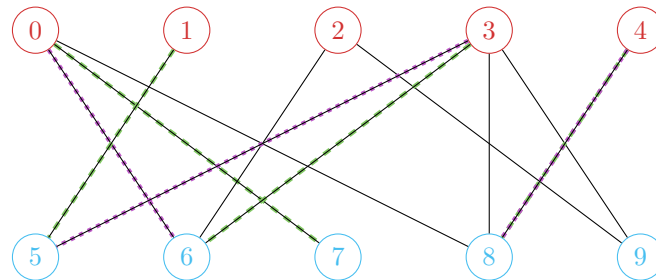
- $K_{3,3}$
- $K_{n,p}$
- les arbres
- les graphes 2-coloriables (c'est même une caractérisation des graphes bipartis)
- les graphes de préférence, ou de mariages

2.1.1 Couplage dans un graphe biparti

Définition 2.13 (Couplage). Un *couplage* M est un sous-ensemble de E tel que

$$\forall v \in M, \quad \#\{e \in M : v \in e\} \leq 1.$$

Dit autrement, M est un couplage lorsque $M \subset E$ et le graphe $G' = (V, M)$ est constitué de sommets isolés ou de segments.



Il y a deux couplages :

- $(0, 6), (3, 5), (4, 8)$ maximal pour l'inclusion (en points violet),
- $(0, 7), (1, 5), (3, 6), (4, 8)$ maximal pour le cardinal (en tirets verts).

Trouver un couplage maximal (pour l'inclusion) est facile : on considère chaque arête, si ses deux extrémités sont des sommets libres (c'est-à-dire pas dans le couplage), on ajoute l'arête au couplage (il s'agit d'un algorithme glouton). Trouver un couplage maximum (pour le cardinal) est bien moins facile et motive la suite de cette partie :

Définition 2.14 (Chemin améliorant). On considère un couplage M pour le graphe G . Un *chemin améliorant* est une suite de sommets v_1, \dots, v_n tels que v_1 et v_n sont libres, et pour tout i , $v_{2i+1} \sim v_{2i+2}$ est une arête de $E \setminus M$ et $v_{2i} \sim v_{2i+1}$ est une arête de M .

Remarque. On notera que n est nécessairement pair, puisque v_n est le bout d'une arête de $E \setminus M$ (sans quoi il ne serait pas libre).

Définition 2.15 (Différence symétrique).

$$A \oplus B := (A \cup B) \setminus (A \cap B) = (A \setminus B) \sqcup (B \setminus A).$$

Remarque. On voit aussitôt le lien entre la différence symétrique et l'opérateur XOR.

Proposition 2.16. Soit M un couplage de G . Si π est un chemin améliorant pour M alors (en confondant π avec ses arêtes), $\pi \oplus M$ est un couplage de G de cardinal $\#M + 1$.

Démonstration. Si π est formé dans l'ordre des arêtes $v_1 \sim v_2, v_2 \sim v_3, \dots, v_{n-1} \sim v_n$. π est un chemin améliorant donc v_n est libre donc $v_{n-1} \sim v_n \in M$. Donc n est pair. On a $\#(\pi \cap M) = \frac{n}{2} - 1$ et $\#(\pi \setminus M) = \frac{n}{2}$ (par récurrence sur $p := \frac{n}{2}$). Finalement, on a

$$\#(\pi \oplus M) = \#M - \left(\frac{n}{2} - 1\right) + \frac{n}{2} = \#M + 1. \quad \blacksquare$$

On lance un quasi-parcours en profondeur à partir des sommets libres qui alterne arête libre et arête couplée et qui s'arrête sur un sommet libre.

Théorème 2.17. M est de cardinal maximum si, et seulement si, il n'existe aucun chemin améliorant.

Démonstration. S'il existe un chemin améliorant, alors d'après 2.16, le couplage n'est pas maximum.

On se donne un couplage M non maximal et un couplage maximum M_{opt} . Posons $N := M \oplus M_{\text{opt}}$. N est un ensemble d'arêtes. Soit $G_N := (V, N)$. $\forall v \in V$, $d^o(v) \leq 2$ dans G_N car $d^o(v)$ dans (V, M) est au plus 1 et $d^o(v)$ dans le graphe V, M_{opt} est au plus 1 car M et M_{opt} sont des couplages et $N \subset M \cup M_{\text{opt}}$. Une composante connexe d'un graphe où tous les sommets sont de degré au plus 2 est

1. soit un sommet isolé,
2. soit un chemin élémentaire,
3. soit un cycle élémentaire.

Dans les cas 2 et 3, sur chaque chemin alterne une arête de M et une arête de M_{opt} . N n'est pas composé uniquement de sommets isolés ou de cycles élémentaires car un cycle élémentaire on trouve autant d'arête de M que de M_{opt} donc $\#M = \#M_{\text{opt}}$ ce qui est absurde. S'il existe un chemin élémentaire $v_1 \sim v_2 \sim \dots \sim v_{n-1} \sim v_n$ en supposant que $v_1 \sim v_2 \in M_{\text{opt}}$ et $v_{n-1} \sim v_n \in M_{\text{opt}}$ alors c'est un chemin améliorant. Sinon, tous les chemins élémentaires maximaux sont de la forme

1. $v_1 \sim v_2 \sim \dots \sim v_{n-1} \sim v_n$ où $v_1 \sim v_2 \in M_{\text{opt}}$ et $v_{n-1} \sim v_n \in M$ a autant d'arêtes de M que de M_{opt} ;
2. $v_1 \sim v_2 \sim \dots \sim v_{n-1} \sim v_n$ où $v_1 \sim v_2, v_{n-1} \sim v_n \in M$ ce qui est absurde car ce serait un chemin améliorant pour M_{opt} ;
3. $v_1 \sim v_2 \sim \dots \sim v_{n-1} \sim v_n$ où $v_1 \sim v_2 \in M$ et $v_{n-1} \sim v_n \in M_{\text{opt}}$ a autant d'arêtes de M que de M_{opt} . \blacksquare

si v_1 est libre si `chemin_améliorant(v1)` renvoie (v_2, \dots, v_n) alors (v_1, \dots, v_n) est un chemin améliorant
fin si

En supposant que \oplus est en $O(1)$ (ce qui est faux en pratique), cet algorithme s'exécute en temps

$$O\left(\underbrace{\#V}_{\text{un couplage contient au plus } \#V/2 \text{ arêtes}} \left(\underbrace{\#E + \#V}_{\text{temps du parcours}} \right)\right)$$

Algorithme 2.2 Trouver un chemin améliorant

Entrée : Graphe $G = (V, E)$ **Sortie :** Un chemin améliorant**fonction** CHEMIN_AMÉLIORANT(v) : **Si** v a déjà été vu **alors** **renvoyer** \emptyset

▷ Échec

Fin si Marque v comme étant vu **Pour** s voisin de v , **faire** : **Si** s est libre **alors** **renvoyer** $[s]$ **Fin si** **Si** s est couplé à t **alors** $\pi \leftarrow \text{CHEMIN_AMÉLIORANT}(t)$ **Si non** $\pi = \emptyset$ **alors** **renvoyer** $s :: t :: \pi$ ▷ On ajoute s et t au chemin π **Fin si** **Fin si** **Fin pour** **renvoyer** \emptyset **Fin fonction**

Algorithme 2.3 Trouver le couplage de plus grand cardinal

 $M \leftarrow \emptyset$ **Tant que** il existe un chemin améliorant π , **faire** : $M \leftarrow M \oplus \pi$ **Fin tant que**

Pour la culture. Les résultats précédents ne sont valides que sur un graphe bipartis. La différence, implicitement utilisée, est que dans un graphe qui n'est pas biparti on peut trouver des cycles de longueurs impairs ce qui fait rater la disjonction de cas. Il faut donc distinguer les chemins alternants de ceux qui ne le sont pas dans les chemins améliorants...

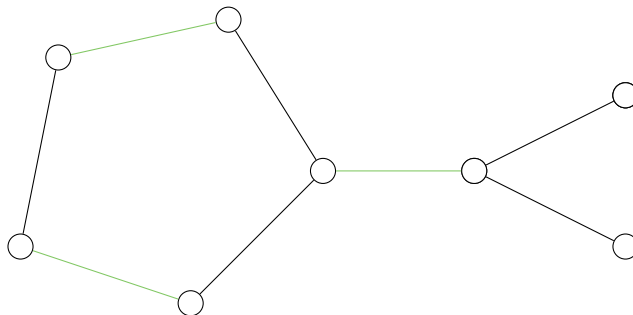


FIGURE 2.2 – Un graphe cerf volant

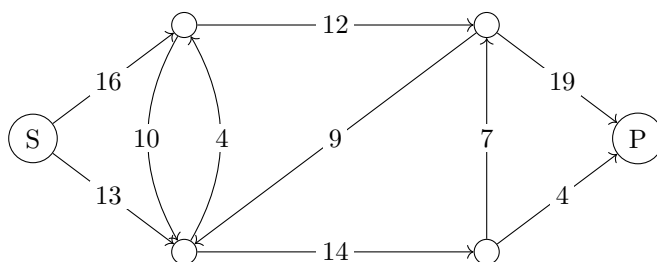
Un couplage possible a été représenté en vert dans le graphe précédent.

Remarque. On appelle ce genre de sous-graphe des *fleurs* (cycle de longueur impair $2k + 1$ avec k arêtes dans le couplage).

où l'algorithme ne pourra pas être correct bien que le reste de la preuve soit encore valide. Le Théorème 2.17 reste alors vrai. Des algorithmes plus généraux existent comme *blossom*. De manière générale, l'informaticien saura toujours déterminer des couplages dans des graphes de nature quelconque.

2.1.2 Lien aux flots dans les graphes (HP)

Une autre extension possible est le couplage de poids maximum (dont la pénibilité est certaine). On considère que G est pondéré par $w : E \rightarrow \mathbb{N}$. Le poids d'un couplage M c'est $\sum_{e \in M} w(e)$. Pour résoudre ce problème, il existe l'algorithme Hongrois (voir Mines MP 2012). On peut appliquer ces méthodes pour résoudre le flot maximum dans un graphe. Dans un réseau de tuyaux de diamètres différents s'écoulent de l'eau, on suppose qu'à chaque intersection toute l'eau qui entre ressort, quelle quantité sort du réseau ? On peut voir la situation par un graphe non orienté (ou non, les deux versions existent) dont les arêtes sont pondérées par des entiers $c : V \times V \rightarrow \mathbb{N}$. Le sommet duquel émerge le flot est appelé source s , le sommet vers lequel s'écoule le flot est appelé puits p .



Voir *Cormen* au chapitre *Flot maximum*.

Définition 2.18. Un flot est une fonction antisymétrique $f : V \times V \rightarrow \mathbb{Z}$ telle que

$$\forall (u, v) \in V^2, \quad f(u, v) \leq c(u, v).$$

On remarque alors que,

$$\forall u \in V, \quad \sum_{v \in V} f(v, u) = 0 = \sum_{v \in V} f(u, v),$$

à l'exception du puits et de la source du graphe peut être.

Définition 2.19. (valeur d'un flot) On appelle valeur d'un flot f l'entier

$$|f| := \sum_{v \in V} f(s, v).$$

Exercice. Montrer que

$$|f| = \sum_{v \in V} f(v, p).$$

Solution :

$$|f| := \sum_{v \in V} f(s, v) = \sum_{v \in V} f(s, v) + \sum_{w \in V \setminus \{s\}} \sum_{v \in V} f(w, v)$$

Le problème consiste alors à trouver un flot de valeur maximum

Exemple. Dans la graphe précédent, avec le flot

on trouve le flot maximum valant 23.

Un algorithme pour calculer un flot maximum utilise également un chemin améliorant en considérant le graphe des résiduels.

Définition 2.20 (Graphe des résiduels). Le graphe des résiduels de G est le graphe dont les capacités sont les capacités actuelles auxquelles on retranche le flot le traversant.

Définition 2.21. Un chemin améliorant entre $(s, t) \in V^2$ est un chemin de poids minimum dans le graphe des résiduels de G .

On peut alors démontrer un théorème similaire

Théorème 2.22. Lorsque l'on possède un flot maximum, il n'y a pas de chemin améliorant.

On exploite ce dernier dans l'algorithme de []. Un chemin améliorant se trouve de nouveau en faisant un parcours dans le graphe des résiduels.

Définition 2.23 (Coupure dans un graphe connexe). Soit $G := (V, E)$ un graphe connexe dont on se donne deux sommets s et p . Une coupure de G est un ensemble d'arêtes C tel que le graphe $(V, E \setminus C)$ sépare s et p en deux composantes connexes.

Le coût d'une coupure C se définit par

$$|C| := \sum_{e \in C} c(e).$$

Théorème 2.24. Le flot maximum est le coût minimum d'une coupure.

2.2 Graphes orientés acycliques (DAG)

Définition 2.25. Un DAG (*Directed Acyclic Graph*) est un graphe orienté qui ne contient pas de cycle.

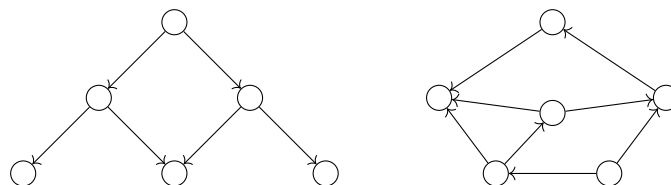


FIGURE 2.3 – Quelques DAG

Proposition 2.26. Si G est un DAG, alors il existe un sommet sans arête entrante.

Démonstration. On raisonne par l'absurde. Soit $v_0 \in V$ possédant, par hypothèse, une arête entrante $v_1 \rightarrow v_0$. Par récurrence on construit une suite $(v_n)_{n \in \mathbb{N}}$ telle que

$$\forall i \in \mathbb{N}, \quad v_{i+1} \rightarrow v_i.$$

Or notre graphe est fini donc par principe de Dirichlet (faisons chic!) il existe deux entiers i et j distincts tels que $v_i = v_j$. On vient de mettre en évidence un cycle orienté ce qui achève la preuve. ■

2.2.1 Tri topologique

Définition 2.27 (Tri topologique). Soit G un DAG. Un tri topologique est une énumération des sommets v_1, \dots, v_n qui vérifie

$$\forall (i, j) \in \llbracket 1, n \rrbracket^2, \quad v_i \rightarrow v_j \implies i < j$$

Proposition 2.28. *Il existe toujours un tri topologique.*

On propose une preuve algorithmique :

Démonstration. Un sommet sans arête entrante convient pour v_1 . On retire alors v_1 du graphe et on réitère avec ce nouveau graphe qui est toujours orienté et acyclique d'où une construction naturelle par récurrence. ■

Remarque. L'algorithme précédent s'évalue en $O(|V|^2 + |E|)$, c'est-à-dire en $O(|V|^2)$ puisque $|E| \leq |V|^2$.

Pour calculer un tri topologique, on fait un parcours en profondeur où l'on empile les sommets dont on a visité tout les voisins.

Algorithme 2.4 Recherche (bis) d'un tri topologique

Entrée : Graphe orienté acyclique $G = (V, E)$

Sortie : Un tri topologique

```

fonction DFS(v,pile) :
    Si seen(v) alors
        renvoyer
    Fin si
    seen(v)  $\leftarrow \top$ 
    Pour chaque voisin s de v, faire :
        DFS(s)
    Fin pour
    EMPILER(v,pile)
Fin fonction

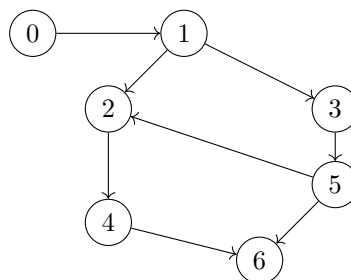
```

```

fonction TRI_TOPOLOGIQUE(G) :
    pile  $\leftarrow \emptyset$ 
    Pour chaque  $v \in V$ , faire :
        seen(v)  $\leftarrow \perp$ 
    Fin pour
    Pour chaque  $v \in V$ , faire :
        DFS(v, pile)
    Fin pour
Fin fonction

```

Exemple. Considérons le graphe suivant



on obtient

Une application du tri topologique est, pour rappel, le makefile.

2.2.2 Composantes fortement connexes

Dans un graphe orienté, la relation \sim sur les sommets définie par $u \sim v$ s'il existe un chemin de u à v et un chemin de v à u est une relation d'équivalence (on a forcé la symétrie de la relation en corrigeant son défaut causé par l'orientation des arêtes).

Définition 2.29. Soit G un graphe orienté. Les classes d'équivalences de \sim sont appelées composantes fortement connexes de G .

Exemple. Considérons le graphe suivant

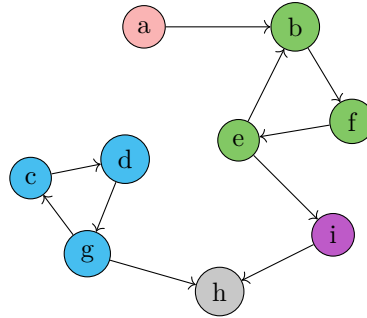


FIGURE 2.4 – Un graphe orienté et ses composantes fortement connexes (colorées)

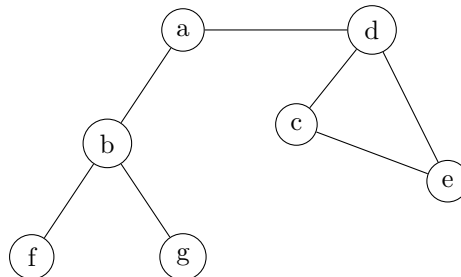
Définition 2.30 (Graphe transposé). Soit $G := (V, E)$ un graphe orienté. On appelle graphe transposé de G , et noté G^T le graphe d'ensemble de sommet V et dont l'ensemble des arêtes est $\{(y, x)\}_{(x, y) \in E}$

Remarque. Ce graphe est donc obtenu en retournant arêtes du graphe initial. Sa dénomination est dû au fait que pour obtenir la matrice d'adjacence de G^T , il suffit de transposer celle de G au sens matriciel.

Définition 2.31. Lors d'un parcours en profondeur, pour chaque sommet u , on note

- $d(u)$ le temps où le parcours découvre u pour la première fois,
- $f(u)$ le temps où le parcours fini de traiter u , c'est-à-dire quand tous les voisins de u sont traités.

Exemple. Considérons le graphe :



On a :

u	$d(u)$	$f(u)$
a	0	13
b	1	6
c	8	11
d	7	12
e	9	10
f	2	3
g	4	5

Question évidente de l'informaticien, existe-t-il un algorithme de calcul de ces composantes fortement connexes ? Il en existe en effet plusieurs : celui de TARJAN ou celui de KOSARAJU que voici :

- On lance un parcours en profondeur du graphe, on note pour chaque sommet u les temps $f(u)$,
- On calcule le graphe transposé de G ,
- On lance un parcours en profondeur depuis tous les sommets par ordre de f décroissante,

Tous les sommets découverts depuis un même sommet de départ forment exactement une composante fortement connexe de G .

Algorithme 2.5 Algorithme de KOSARAJU

Entrée : Graphe $G = (V, E)$

Sortie : le nombre des composantes fortement connexes et un tableau de ces composantes

```

fonction DFS( $v$ ) :
  Si  $\neg visited[v]$  alors
     $visited[v] \leftarrow \top$ 
     $num[v] \leftarrow nc$ 
    Pour  $v$  sommet voisin de  $g$ , faire :
      DFS( $v$ )
    Fin pour
  Fin si
Fin fonction

fonction COMPONENT( $v$ ) :
  Si  $\neg visited[v]$  alors
    renvoyer DFS( $v$ ), incr  $nc$ 
  Fin si
Fin fonction

fonction KOSARAJU( $G$ ) :
   $n \leftarrow |g|$ 
   $nc \leftarrow 0$ 
   $num \leftarrow$  tableau de  $n$  0
   $visited \leftarrow$  tableau de  $n$   $\perp$ 
  Pour  $v$  sommet de l'ordre décroissant du miroir de  $g$ , faire :
    COMPONENT( $v$ )
  Fin pour
  renvoyer  $nc$ ,  $num$ 
Fin fonction

```

Exemple. Reprenons le graphe 2.2.2 sur lequel on applique l'algorithme de KOSARAJU. On choisit comme sommet de départ du parcours en profondeur le sommet f et on obtient le tableau :

u	$d(u)$	$f(u)$
a	10	11
b	6	7
c	14	15
d	12	17
e	1	8
f	0	9
g	13	16
h	3	4
i	2	5

Le deuxième parcours dans G^T déroule

- $d \rightarrow c, g$
- a
- $f \rightarrow b, e$

- i
- h

et l'on retrouve les composantes connexes de la figure.

On cherche donc désormais à montrer la correction de l'algorithme de KOSARAJU (sachant la terminaison immédiate, on effectue deux parcours en profondeurs).

Proposition 2.32. *G et G^T ont les mêmes composantes connexes.*

Démonstration. On note $\mathcal{C}(K)$ l'ensemble des chemins d'un graphe K . Considérons l'application :

$$\begin{array}{ccc} \varphi : & \mathcal{C}(G) & \longrightarrow \mathcal{C}(G^T) \\ & (u_1 \rightarrow \dots \rightarrow u_n) & \longmapsto (u_n \rightarrow \dots \rightarrow u_1) \end{array}$$

qui est bien définie par construction du graphe G^T . Et de même considérons :

$$\begin{array}{ccc} \psi : & \mathcal{C}(G^T) & \longrightarrow \mathcal{C}(G) \\ & (u_1 \rightarrow \dots \rightarrow u_n) & \longmapsto (u_n \rightarrow \dots \rightarrow u_1) \end{array}$$

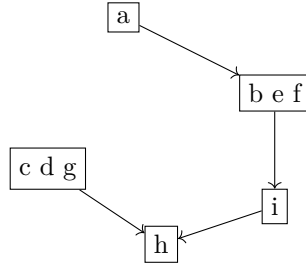
qui est aussi bien définie. Les relations suivantes viennent alors naturellement : $\varphi \circ \psi = \text{Id}_{\mathcal{C}(G)}$ puis $\psi \circ \varphi = \text{Id}_{\mathcal{C}(G^T)}$. Ainsi φ et ψ sont bijectives réciproques l'une de l'autre (autrement dit, on a formalisé l'idée que tout chemin de G^T s'obtient en retournant un chemin de G). Par suite en appliquant φ et ψ on obtient la condition

$$C \text{ est une composante connexe de } G \iff C \text{ est une composante connexe de } G^T,$$

d'où le résultat. ■

Définition 2.33 (G^{SCC}). Le graphe des composantes connexes G^{SCC} est un graphe dont les composantes fortement connexes de G sont les sommets. De plus, on décrète qu'il y a une arête $C \rightarrow C'$ si et seulement s'il existe $(u, v) \in C \times C'$ tel que $u \rightarrow v$ est une arête de G

Remarque. SCC veut dire *Strongly Connected Components*.



Proposition 2.34. *Le graphe des composantes connexes G^{SCC} est un DAG*

Démonstration. On raisonne par l'absurde, supposons qu'il existe deux composantes fortement connexes distinctes C et C' telles que

$$C \rightarrow C' \quad \text{et} \quad C' \rightsquigarrow C.$$

La seconde hypothèse donne donc l'existence de $(C_i)_{1 \leq i \leq p}$ des composantes fortement connexes telles que

$$C' := C_1 \rightarrow \dots \rightarrow C_p =: C,$$

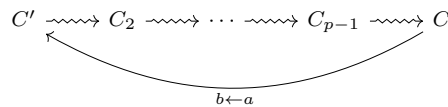
dans G^{SCC} . Par suite, pour tout $i \in \llbracket 1, p-1 \rrbracket$, il existe $(u_i, v_{i+1}) \in C_i \times C_{i+1}$ tel que $u_i \rightarrow v_{i+1}$ dans G . Or, pour $i \in \llbracket 2, p-1 \rrbracket$ donné, par forte connexité de C_i on a $v_i \rightsquigarrow u_i$. On dispose ainsi d'un chemin

$$u_1 \rightarrow v_2 \rightsquigarrow u_2 \rightarrow \dots \rightarrow v_j \rightsquigarrow u_j \rightarrow \dots \rightarrow v_{p-1} \rightsquigarrow u_{p-1} \rightarrow v_p$$

c'est à dire d'un chemin $u_1 \rightsquigarrow v_p$ entre C_1 et C_p c'est à dire entre C' et C (en ayant composé les chemins ci-dessus). Or étant donné que $C \rightarrow C'$, il existe $(a, b) \in C \times C'$ tel que $a \rightarrow b$. Par forte connexité de C et C' on obtient alors

$$b \rightsquigarrow u_1 \rightsquigarrow v_p \rightsquigarrow a \rightarrow b$$

que l'on peut visualiser comme ceci



D'où l'existence d'un cycle permettant d'affirmer que u et v sont dans la même composante fortement connexe par transitivité de la relation \rightsquigarrow . Par conséquent $C = C'$ c'est absurde. ■

Proposition 2.35. *Si C est une composante fortement connexe, on note :*

$$d(C) := \min_{v \in C} d(v) \quad \text{et} \quad f(C) := \max_{v \in C} f(v)$$

S'il existe une arête $C \rightarrow C'$ dans G^{SCC} , alors $f(C) > f(C')$

Démonstration. On distingue 2 cas :

- Si $d(C) < d(C')$, on note v le premier sommet visité dans C . Alors tous les sommets de C et tous les sommets de C' sont non visités lorsque l'on découvre v . Si on considère $u \in C' \cup C$ alors il existe un chemin de v à u constitué uniquement des sommets non visités donc tout sommet de $u \in C' \cup C$ est découverte par cette parcours depuis v . Donc $\forall u \in C' \cup C \setminus \{v\}, f(v) > f(u)$. Donc $f(C) > f(C')$.
- Si $d(C) > d(C')$, on note u le premier sommet visité dans C' . Alors, d'après le résultat précédant, le parcours depuis u n'atteint aucun sommet de C . A la fin de parcours, tous les sommets de C' sont visités et non ceux de C de sorte que :

$$\forall w \in C', \quad d(u) \leq d(w) \leq f(w) \leq f(u) \leq d(c),$$

donc $f(C') < f(C)$. ■

L'idée générale de l'analyse qui vient d'être réalisée fait remarquer la possibilité d'isoler un sommet dont la composante fortement connexe sera vue en dernière et ceci laisse la possibilité d'un raisonnement récursif.

Théorème 2.36. *L'algorithme de Kosaraju est correct.*

Démonstration. Dans G le sommet v telle que $f(v)$ est maximum appartient à une composante fortement connexe C qui, d'après le lemme ci-dessus, n'as pas d'arêtes entrantes dans G^{SCC} . Donc dans G^T si $u \in C$, et si $u \rightarrow v \in G^T$ alors u et v sont dans C . En effet, une arête $u \rightarrow v$ dans G^T donne une arête $v \rightarrow u$ dans G et par conséquent dans G^{SCC} :

- Soit $C' \rightarrow C$, ce qui est exclu par maximalité de $f(C)$,
- Soit $C \rightarrow C$ qui est donc retenu.

On peut alors conclure par récurrence sur le nombre de composante fortement connexes :

- S'il n'y a pas de composantes fortement connexe, c'est immédiat.
- Si le résultat est acquis pour tout graphe à $k \in \mathbb{N}$ composantes connexes. Soit G un graphe à $k + 1$ composantes connexes. On se donne un sommet v comme précédemment. Le parcours dans G^{SCC} depuis v découvre C et uniquement C (c'est le résultat précédant). Par conséquent, l'algorithme de KOSARAJU exécutera DFS sur un sommet qui n'appartient pas à C ce qui revient au même que d'exécuter ce dernier algorithme sur le graphe G^T duquel on a retiré la composante fortement connexe C et on sait alors par hypothèse de récurrence que l'algorithme de KOSARAJU trouvera les k dernières composantes fortement connexe et donc toutes les composantes fortement connexes auront été découvertes ce qu'on voulait montrer. ■

2.2.3 2-SAT

Une formule en entrée de 2-SAT est une formule en Forme Normale Conjonctive (FNC) où chaque clause contient deux littéraux. (On appelle ça une 2-FNC). 2-SAT est le problème de décision qui à chaque 2-FNC associe le booléen qui indique si la formule est satisfiable, c'est-à-dire un algorithme qui résout le problème 2-SAT, on dit qu'il décide si une 2-FNC est satisfiable.

Chaque clause de la forme $x \vee y$ est équivalente (en logique classique) à $\neg x \Rightarrow y$ et à $\neg y \Rightarrow x$. On construit le graphe dont les sommets sont tous les littéraux possibles, et où l'on place une arête $l_1 \rightarrow l_2$ si l'une des clauses de la formule est équivalente à $l_1 \Rightarrow l_2$. Dans ce graphe,

- si l est un littéral tel que l et $\neg l$ ne sont pas dans la même composante fortement connexe, On met à vrai tous les littéraux de sa composante fortement connexe et on jette la composante qui contient l ce qui permet d'obtenir un témoin de satisfiabilité.
- si l et $\neg l$ sont dans la même composante fortement connexe, c'est donc que $l \iff \neg l$ qui n'est pas satisfiable.

Les deux tirets ci-dessus forment une condition nécessaire et suffisant garantissant la résolution de 2-SAT. Par suite, la résolution de la 2-FNC en graphe se fait-elle en temps polynomial? Si $G := (V, E)$ est le graphe produit depuis F On note n le nombre de variables de F et m le nombre de clauses de F . On a $|F| = 2m$ puisque chaque clause contient deux littéraux et $n \leq |F|$ (Chaque variable apparaît au moins une fois). A toute variable, on peut attribuer deux littéraux donc $|V| = 2n$. La construction de V se fait

donc en temps $O(n)$, donc en $O(|F|)$. Puis, la construction de E se fait en parcourant F et en ajoutant 2 arêtes par clause, donc en temps $O(m) = O(|F|)$ de sorte que finalement G est de taille $O(|F|)$. Ainsi, On peut décider 2-SAT en temps polynomial.

Exemple. Considérons :

$$F := (a \vee b) \wedge (\neg d \vee \neg c) \wedge (\neg a \vee b) \wedge (\neg b \vee d).$$

On obtient le graphe orienté suivant :

dont il apparaît, d'après l'analyse ci-dessus, que F est satisfiable.

2.3 Arbres couvrants

Un arbre couvrant est un ensemble d'arêtes qui forment un arbre contenant tous les sommets. L'avantage de l'arbre est qu'il existe un unique chemin reliant deux sommets.

Définition 2.37 (Arbre couvrant). Si G est un graphe non orienté, un arbre couvrant est donné par un ensemble d'arêtes $S \subset E$ tel que (V, S) est un arbre

Proposition 2.38. *Si G est connexe, alors il existe un arbre couvrant.*

Démonstration. Il suffit de considérer l'arbre renvoyé par DFS. ■

On peut aussi penser à l'algorithme (qui pourrait faire office de preuve algorithmique à la proposition qui vient d'être démontrée) suivant que l'on analyse dans la suite.

Algorithme 2.6 Algorithme de calcul d'un arbre couvrant (*Union-Find*)

Entrée : Graphe $G = (V, E)$

Sortie : $S \subset E$ tel que (V, S) est un arbre couvrant

fonction ARBRECOUVRANT(E) :

$S \leftarrow \emptyset$

Pour $e := (x, y) \in E$, **faire** :

Si x et y ne sont pas déjà reliés par un chemin **alors**

$S \leftarrow S \sqcup \{e\}$

Fin si

Fin pour

renvoyer S

Fin fonction

Lemme 2.39. (V, S) est acyclique est un invariant de cet algorithme

Démonstration. Si S est acyclique en début de boucle. Si $S' := S \sqcup \{e\}$ où e relie x et y contient un cycle alors ce cycle prend l'arête e donc il existe un chemin de x à y dans S ce qui est absurde car l'algorithme prend e seulement si x et y ne sont pas reliés dans S . Donc S' est acyclique. ■

Lemme 2.40. A la fin de l'algorithme (V, S) est connexe

Démonstration. On raisonne par l'absurde. Si le graphe (V, S) n'est pas connexe, alors il existe une arête $e \in E$ qui relie deux composantes connexes de (V, S) . Or, l'algorithme ne fait qu'ajouter des arêtes à S . Donc la valeur de S' de S au moment où l'algorithme a traité e vérifie $S' \subset S$. Mais, $S \sqcup \{e\}$ est acyclique donc $S' \sqcup \{e\}$ est acyclique donc l'algorithme a ajouté e ce qui est absurde. ■

Proposition 2.41. *Le résultat est un arbre couvrant.*

Démonstration. En vertu des deux lemmes démontrés ci-dessus, le résultat de ARBRECOUVRANT(E) est connexe et acyclique, c'est donc un arbre. Par conséquent, puisque l'ensemble des sommets reste inchangé, ce dernier est couvrant pour G d'où la proposition. ■

Définition 2.42. Si on se donne une fonction $w : E \rightarrow \mathbb{N}$, le *poids* de $S \subset E$ est

$$w(S) := \sum_{e \in S} w(e)$$

La question de l'informaticien, comment obtenir un arbre couvrant de poids minimal ? La réponse est aussi simple que surprenante, on reprend l'algorithme 2.3, mais on impose un ordre lors de la sélection des sommets (par poids croissant). On obtient :

Algorithme 2.7 Algorithme de KRUSKAL

Entrée : Graphe $G = (V, E)$

Sortie : $S \subset E$ tel que (V, S) est un arbre couvrant de poids minimal

fonction KRUSKAL(E) :

$S \leftarrow \emptyset$

$E' \leftarrow \text{TRIER}(E)$

Pour $e := (x, y) \in E'$ dans l'ordre, **faire** :

Si x et y ne sont pas déjà reliés par un chemin **alors**

$S \leftarrow S \sqcup \{e\}$

Fin si

Fin pour

renvoyer S

Fin fonction

dont on peut évaluer la complexité, si $m := \#E$ alors :

— Trier les arêtes se fait en $O(m \log(m))$.

— L'intérieur de la boucle se fait en $O(\alpha(m))$ amorti avec *Union Find*, donc la boucle se fait en $O(m\alpha(m))$.

Or, $\alpha = o(\log)$ (où α est la réciproque de la fonction d'ACKERMANN) donc la composition est en $O(m \log(m))$. On s'intéresse désormais à la correction de l'algorithme de KRUSKAL qui repose essentiellement sur le lemme suivant :

Lemme 2.43. Si e_0 est l'arête de poids minimal considérée en premier par l'algorithme, il existe une solution optimale qui contient e_0 comme arête.

Démonstration. Soit $\mathcal{T} := (S, T)$ un arbre couvrant de poids minimum

— Si $e_0 \in T$, il n'y a rien à montrer.

— Si $e_0 \notin T$. Soient x et y les sommets reliés par e_0 . Puisque \mathcal{T} est un arbre, il existe un chemin de x à y formé d'arêtes e_1, \dots, e_n dans \mathcal{T} . Posons $\mathcal{T}' := (S, (T \setminus \{e_1\}) \sqcup \{e_0\})$.

Soient u, v deux sommets. Dans \mathcal{T} , u et v sont reliés par un unique chemin élémentaire.

— Si ce chemin ne passe pas par e_1 , alors u et v sont reliés par le même chemin dans \mathcal{T}' .

— Si ce chemin passe par e_1 , on remplace dans ce chemin l'arête e_1 par le chemin $(e_2, e_3, \dots, e_n, e_0)$. C'est un chemin de u à v dans \mathcal{T}' .

Par conséquent, \mathcal{T}' est connexe et puisque par construction :

$$\#T = \#T', \quad |\mathcal{T}| = |\mathcal{T}'| \quad \text{donc} \quad \#T' = |\mathcal{T}'| - 1,$$

on déduit, par caractérisation des arbres 2.11, que \mathcal{T}' est un arbre couvrant. Enfin, on a $w(T') = w(T) - w(e_1) + w(e_0)$. Or, $w(e_0)$ est minimal, donc $-w(e_1) + w(e_0) \leq 0$ de sorte que $w(T') \leq w(T)$ si bien que, étant donné l'optimalité de \mathcal{T} :

$$w(T) \leq w(T') \leq w(T) \quad \text{donc} \quad w(T) = w(T')$$

d'où l'optimalité de \mathcal{T}' ce qu'on voulait obtenir. ■

Théorème 2.44. L'algorithme de KRUSKAL est correct

Démonstration. En premier lieu, il convient de remarquer que le résultat est un arbre couvrant car l'algorithme 2.3 (sans les poids) est correct quel que soit l'ordre sur les arêtes d'après 2.41. Il nous reste cependant un dernier point à vérifier. Par récurrence forte sur le nombre d'arêtes à examiner (ce qui utilise indirectement un principe sur les matroïdes que l'on étudie au chapitre hors programme XXX couplé au lemme montré juste avant), l'algorithme de KRUSKAL trouve une solution optimale parmi celles qui commencent par e_0 . ■

2.4 Distances dans les graphes

2.4.1 Définitions

Soit $(u, v) \in V^2$. On note $\Gamma_{u,v}$ l'ensemble des chemins de u à v et $\Gamma_{u,v}^e$ l'ensemble des chemins élémentaires de $\Gamma_{u,v}$. Dans un graphe (orienté ou non), on se donne une fonction $w : E \rightarrow \mathbb{Z}$ (ou \mathbb{N}). Si γ est un chemin qui passe par les arêtes (e_1, \dots, e_n) alors, on définit :

$$w(\gamma) := \sum_{i=1}^n w(e_i).$$

Définition 2.45 (Distance). On définit la *distance entre deux sommets* comme

$$d : V^2 \longrightarrow \mathbb{Z}$$

$$(u, v) \longmapsto \begin{cases} \min\{w(\gamma)\}_{\gamma \in \Gamma_{u,v}} & \text{si } \Gamma_{u,v} \neq \emptyset \\ \infty & \text{sinon} \end{cases}$$

Proposition 2.46. d est définie si et seulement si le graphe n'a pas de cycle de poids négatif.

Démonstration. On suppose que le graphe est orienté (mais la preuve est la même si il ne l'est pas) et n'a pas de cycle de poids strictement négatif. Soit $(u, v) \in V^2$. Posons

$$A := \{w(\gamma)\}_{\gamma \in \Gamma_{u,v}} \quad \text{et} \quad B := \{w(\gamma)\}_{\gamma \in \Gamma_{u,v}^e}.$$

Puisque clairement $\Gamma_{u,v}^e \subseteq \Gamma_{u,v}$, alors $B \subseteq A$ si bien que $l_A := \inf(A) \leq \inf(B) =: l_B$ (les bornes inférieures sont prises dans \mathbb{R} et par conséquent existent). Montrons par récurrence forte sur la longueur d'un chemin que

$$\forall \gamma \in \Gamma_{u,v}, \quad \exists \gamma' \in \Gamma_{u,v}^e, \quad w(\gamma') \leq w(\gamma).$$

- Lorsque la longueur d'un chemin est de longueur inférieure à 3, le résultat désiré est immédiat.
- Sinon soit $n \in \mathbb{N}$ tel que pour tout chemin de longueur plus petite que n , la propriété à démontrer soit vérifiée. Soit alors $\gamma := (v_1, \dots, v_{n+1}) \in \Gamma_{u,v}$ de longueur $n+1$. Deux cas se distinguent :
 - Si $\gamma \in \Gamma_{u,v}^e$ alors directement $w(\gamma) \geq w(\gamma)$.
 - Sinon, γ admet un cycle dont on note k sa longueur (qui est non nul par ailleurs). Par conséquent, il existe $i \in \llbracket 1, n+1-k \rrbracket$ tel que $v_i = v_{i+k}$ (i.e (v_i, \dots, v_{i+k}) est un cycle). Par suite

$$w(\gamma) = \sum_{l=1}^{n+1} w(v_l) = \sum_{l=1}^i w(v_l) + \sum_{l=i+1}^{i+k} w(v_l) + \sum_{l=i+k+1}^n w(v_l) \geq \sum_{l=1}^i w(v_l) + \sum_{l=i+k+1}^n w(v_l),$$

car le poids d'un cycle est positif. Posons alors $\bar{\gamma} := (v_1, \dots, v_i, v_{i+k+1}, \dots, v_n) \in \Gamma_{u,v}$ (clairement) qui est tel que $|\bar{\gamma}| < n+1$ et $w(\gamma) \geq w(\bar{\gamma})$. L'hypothèse de récurrence s'applique alors à $\bar{\gamma}$ et donne l'existence de $\gamma' \in \Gamma_{u,v}^e$ tel que $w(\bar{\gamma}) \geq w(\gamma')$, si bien que finalement

$$w(\gamma) \geq w(\bar{\gamma}) \geq w(\gamma'),$$

donc $w(\gamma) \geq w(\gamma')$, ce qui correspond au résultat à montrer au rang $n+1$.

On en déduit alors la validité de la propriété du principe de récurrence. Mais celle-ci entraîne directement que

$$\forall \gamma \in \Gamma_{u,v}, \quad w(\gamma) \geq l_B,$$

et un passage à la borne inférieure donne alors que $l_B \leq l_A$ si bien que *a fortiori* $l_B = l_A$. Or $\Gamma_{u,v}^e$ est une partie finie par principe des tiroirs donc B est une partie finie de \mathbb{Z} et par conséquent l_B est un minimum et comme $l_A = l_B$ alors $\min(A)$ existe d'où découle la bonne définition de d .

La réciproque se traite par contraposé et est évidente. ■

Exemple. Exemples de recherche de chemin le ou les plus courts :

$$\begin{array}{ll} 1 \text{ à } 1 & (\text{Nan}) \\ 1 \text{ à } n & (\text{DIJKSTRA}) \\ n \text{ à } n & (\text{FLOYD-WARSHALL}) \end{array}$$

Remarque. DIJKSTRA calcule les distances d'un sommet à tous les autres uniquement si tous les poids sont positifs.

FLOYD-WARSHALL calcule la matrice des distances entre tous les sommets dans tous les cas.

Algorithme 2.8 Algorithme de DIJKSTRA v.1

$d(s) \leftarrow 0$
 $\forall u \in S - \{s\}, d(u) \leftarrow +\infty$
A chaque étape, parmi les sommets u tels que $d(u) \neq +\infty$ on cherche l'arête (u, v) telle que $d(u) + w(u, v)$ est minimale et que $d(v) = +\infty$
On peut prouver que $d(v) = d(u) + w(u, v)$ donc on écrit $d(v) \leftarrow d(u) + w(u, v)$

2.4.2 Dijkstra

Dijkstra v.1

On a un tableau d qui à chaque sommet associe la meilleure distance connue depuis s ou $+\infty$ si cette distance n'est pas connue.

Montrons que π' est de poids minimal. On sait que π est optimal. Un autre chemin de s jusqu'au sommet u' tel que $d(u')$ est connue, suivi d'un chemin π''' de u à v . Par définition de u et de v , $d(u) + w(u, v)$ est minimale, donc $d(u') + w(u', \dots) \geq d(u) + w(u, v)$

Donc $w(\pi'' \circ \pi''') \geq d(u) + w(u, v)$ car $w(\pi''') \geq 0$

Dijkstra v.2

On se donne une file de priorité qui stocke les priorités, $prio(v) = d(u) + w(u, v)$ pour des v tels que $d(v) = \infty$ et voisins d'un u tel que $d(u) \neq \infty$.

Algorithme 2.9 Algorithme de DIJKSTRA v.2

fonction RELAXER(u) :
 Pour toute arête (u, v) , **faire** :
 Si $v \notin prio$ ou $prio(v) > d(u) + w(u, v)$ **alors**
 $prio(v) \leftarrow d(u) + w(u, v)$
 Fin si
 Fin pour
Fin fonction

fonction DIJKSTRA2(s) :
 $d(u) \leftarrow 0$ si $u = s$, $+\infty$ sinon
 $prio \leftarrow \emptyset$
 relaxer(s)
 Tant que $prio$ n'est pas vide, **faire** :
 $(u, d(u)) \leftarrow \text{POP}(prio)$
 relaxer(u)
 Fin tant que
Fin fonction

Étude du Coût : Si $C(n)$ est le coût de la file de priorité pour changer les priorités ou de pop sur une file à n éléments, on a un DIJKSTRA qui tourne en $(m + n)C(n)$

2.4.3 Floyd-Warshall

On suppose numéroté les sommets de G de 1 à n . Pour tout $k \in \llbracket 1, n \rrbracket$ on note $d_k(u, v)$ la distance de u à v dans le graphe des chemins où tous les sommets intermédiaires sont entre 1 et k noté G_k . Soit u, v deux sommets de G . Observons alors que pour tout $k \in \llbracket 1, n \rrbracket$

- Si $k = 0$ alors $d_0(u, v) = w(u, v)$.
- Sinon $d_k(u, v) = \min\{d_{k-1}(u, k) + d_{k-1}(k, v), d_{k-1}(u, v)\}$.

Car un chemin optimal d'intermédiaire de u à v dans G_k qui :

- soit ne passe pas par le sommet k et donc a pour poids $d_{k-1}(u, v)$.
- soit passe par k , et alors c'est un chemin optimal de u à k concaténé avec un chemin optimal de k à v (si l'un des deux ne l'est pas on contredirait l'optimalité du chemin trouvé), qui ne passent que par des sommets de 1 à $k - 1$ (même argument).

Ceci donne une formule de programmation dynamique qu'il est aisé d'exploiter. Ainsi la correction des algorithmes qui suivent est évidente.

Algorithme 2.10 Algorithme de FLOYD-WARSHALL v.1

```
On crée un tableau à 3 dimensions  $d(k, u, v)$ 
 $d(0, u, v) \leftarrow$  matrice des poids des arêtes
Pour  $k = 1 \rightarrow n$ , faire :
  Pour  $u = 1 \rightarrow n$ , faire :
    Pour  $v = 1 \rightarrow n$ , faire :
       $d(k, u, v) \leftarrow \min\{d(k-1, u, k) + d(k-1, k, v), d(k-1, u, v)\}$ 
    Fin pour
  Fin pour
Fin pour
```

dont la complexité temporelle et spatiale est en $O(n^3)$

Algorithme 2.11 Algorithme de FLOYD-WARSHALL v.2

```
deux tableaux
 $d_{\text{vieux}}(u, v) \leftarrow \text{poids}$ 
 $d_{\text{nouveau}}(u, v)$ 
Pour  $k = 1 \rightarrow n$ , faire :
  Pour  $u = 1 \rightarrow n$ , faire :
    Pour  $v = 1 \rightarrow n$ , faire :
       $d_{\text{nouveau}}(u, v) \leftarrow \min\{d_{\text{vieux}}(u, v), d_{\text{vieux}}(u, k) + d_{\text{vieux}}(k, v)\}$ 
    Fin pour
  Fin pour
  échanger  $d_{\text{nouveau}}$  et  $d_{\text{vieux}}$ 
Fin pour
```

dont la complexité temporelle est $O(n^3)$ et spatiale $O(n^2)$.

Algorithme 2.12 Algorithme de FLOYD-WARSHALL v.3 (c'est lui le "vrai")

```
un tableau
d ← matrice des poids
Pour k = 1 → n, faire :
    Pour u = 1 → n, faire :
        Pour v = 1 → n, faire :
            d(u, v) ← min{d(u, v), d(u, k) + d(k, v)}
        Fin pour
    Fin pour
Fin pour
```

Remarque. Ce troisième algorithme ne respecte pas la formule de programmation dynamique énoncée plus haut, mais est quand même fonctionnel. En effet à chaque instant, $d(u, v)$ contient le poids d'un vrai chemin de u à v et comme c'est un minimum, $d(u, v) \geq d_k(u, v)$. A la fin, on a bien : $d(u, v) = d_n(u, v)$. Au bout de n itérations, $d(i, i) = 0$ si et seulement s'il n'existe pas de cycles de poids strictement négatif.

Ainsi, L'algorithme de FLOYD-WARSHALL stocke les meilleures distances et permet de détecter des cycles de poids strictement négatif.

2.4.4 Retour sur la programmation dynamique

Exemple. On se donne n matrices $(M_1, \dots, M_n) \in \mathcal{M}(\mathbb{K})_{n_1, n_2} \times \dots \times \mathcal{M}(\mathbb{K})_{n_n, n_{n+1}}$ dont on cherche à optimiser le produit $M_1 \times \dots \times M_n$ en temps de calculs. On note $T(i, j)$ = le temps optimal pour calculer le produit $M_i \dots M_j$. Il vient d'ores et déjà :

- $T(i, i) = 0$
- Un produit optimal se décompose en sous-produits optimaux et du produit entre ces derniers, dont il ressort la relation de récurrence :

$$T(i, j) = \min_{0 \leq k \leq j-1} \left\{ T(i, k) + T(k+1, j) + \underbrace{n_i \times n_{k+1} \times n_{j+1}}_{\text{coût de la multiplication}} \right\}.$$

On cherche donc à calculer $T(1, n)$. Remarquons que plusieurs de manières de procéder existent :

- Calculer les $T(i, j)$ par diagonales qui montent.
- Colonne par colonne en remontant depuis la diagonale.

Exemple. L'optimisation de l'espacement des mots sur L^AT_EX est un parfait exemple d'algorithme dynamique, comparé à celui de Word qui utilise un algorithme glouton qui renvoie à la ligne à chaque fois qu'un mot ne rentre pas.

Chapitre 3

Concurrence et parallélisme

Sommaire

3.0	Banalités	47
3.1	Section critique et exclusion mutuelle	48
3.1.1	Algorithme de PETERSON	49
3.2	Mutex à n processus	50
3.2.1	Généralisation de PETERSON (HP)	50
3.2.2	Boulangerie de LAMPORT	50
3.3	Sémaphores	51
3.4	Implémentations	51
3.4.1	En C	52
3.4.2	En OCaml	52
3.5	Quelques exemples au programme	53
3.5.1	Dîner de philosophes	53
3.5.2	Rendez-vous	54
3.5.3	Producteurs-consommateurs	54
3.5.4	Barrière de synchronisation	54
	Exercices	54

Dans ce chapitre, Alice et Bob reviennent. Tout deux étant ennuyés de jouer avec l'autre, ils se décident à vouloir se rendre dans le jardin. Nonobstant, une curieuse raison impose que le jardin ne puisse être réservé que par une personne à la fois¹. Alice et Bob peuvent-ils trouver une stratégie pour mutuellement garantir qu'ils ne se retrouvent pas simultanément au jardin ?

3.0 Banalités

Définition 3.1. Un processus est une exécution en cours d'un programme, ce qui contient :

- Le code du programme
- De la mémoire
- Des informations (droits, états, priorités, etc) pour que le noyau gère le processus.

Dans la mémoire d'un processus, on trouve plusieurs zones :

- La pile
- Le tas
- Du code
- La zone statique
- etc...

1. Dans l'histoire d'origine, chacun veut promener son animal, l'un étant un chat et l'autre un chien

Définition 3.2 (Threads (processus légers, fils d'exécution)). Dans un processus, on se donne plusieurs piles associées à plusieurs extraits de code qui peuvent s'exécuter simultanément

Exemple (Compteur détraqué). On considère un processus qui a deux threads Alice et Bob exécutant le même code, où compteur est une variable globale, partagée entre les deux threads :

```
for (int i = 0; i < 10000; i++) {
    compteur = compteur + 1;
}
```



compteur = compteur + 1 est souvent compilée en 3 instructions :

- LOAD compteur vers un registre 1
- INCR pour augmenter à 2
- STORE 2 dans compteur

Thread	Instruction	compteur
Bob	LOAD	compteur → 0
Alice	LOAD	compteur → 0
Alice	INCR	0 → 1
Alice	STORE	1 → compteur
Bob	INCR	0 → 1
Bob	STORE	1 → compteur

TABLE 3.1 – Exemple d'exécution possible

À la fin, compteur vaut 1

Avec un autre ordre sur les instructions, compteur aurait pu valoir 2.

Le compteur peut prendre à la fin des 2×10000 itérations, les valeurs de 2 à 20000

Il existe différentes formes de concurrence :

- Parallélisme complet (plusieurs coeurs qui s'exécutent simultanément) : p thread
- Concurrency coopérative (explicitement chaque processus léger s'arrête volontairement pour laisser tourner les autres) : c'est présent en Ocaml, Javascript, ou même Python.
- Concurrency non coopérative (l'arrêt est forcé par l'ordonnanceur)

3.1 Section critique et exclusion mutuelle

Définition 3.3. Une section critique est un morceau de code qui ne doit être exécuté que par un seul thread à la fois.

Exemple (Suite compteur détraqué). Dans l'exemple du compteur détraqué, la section critique est l'instruction d'incrémentation du compteur.

Définition 3.4 (Mutex). Le but est d'ajouter une exclusion mutuelle (*mutex*), c'est à dire une structure de données m équipée de deux fonctions : LOCK et UNLOCK, telles que si m est disponible alors LOCK(m) ne termine que pour un seul thread et bloque les autres et UNLOCK(m) par ce thread rend m à nouveau libre pour les threads qui attendaient.

Exemple (Compteur détraqué).

Pour i allant de 1 à 1000, **faire** :

```
    LOCK
    compteur ← compteur + 1
    UNLOCK
```

Fin pour

garantit que compteur vaut 2000 à la fin avec les deux threads.

3.1.1 Algorithme de Peterson

L'algorithme de PETERSON est une implémentation de mutex pour deux threads. Il est intéressant car il n'utilise que des lectures et écritures dans des variables partagées. Notamment, il n'utilise pas d'autres structures de synchronisation pour garantir la synchronisation, il le fait "à partir de rien."

Algorithme 3.1 Algorithme de PETERSON

Entrée : Deux processus A et B

drapeau[A] $\leftarrow \perp$
drapeau[B] $\leftarrow \perp$
victime $\leftarrow \emptyset$

fonction LOCK(X) :
 drapeau[X] $\leftarrow \top$
 victime $\leftarrow X$
 Tant que drapeau[X] **et** victime = X, **faire** :
 ATTENDRE
 Fin tant que
Fin fonction

fonction UNLOCK(X) :
 drapeau[X] $\leftarrow \perp$
Fin fonction

Théorème 3.5 (Correction de PETERSON). *L'algorithme de PETERSON garantit l'exclusion mutuelle, et même l'équité.*

Démonstration. On raisonne par l'absurde. Supposons que Alice (A) et Bob (B) soient tous deux en section critique. On note ici pour une personne X, $READ_X(\cdot)$ lorsque le thread X effectue une opération de lecture en mémoire, $WRITE_X(\cdot)$ lorsqu'il effectue une opération en écriture. On note $E_1 \rightarrow E_2$ lorsque E_1 est réalisée avant E_2 .

Si Alice est entrée en section critique SC_A , alors on a :
 $WRITE_A(drapeau[A] \leftarrow \top) \rightarrow WRITE_A(victime \leftarrow A) \rightarrow READ_A((drapeau[B] \wedge victime = A) = \perp) \rightarrow SC_A$
Si Bob est en section critique SC_B , alors :
 $WRITE_B(drapeau[B] \leftarrow \top) \rightarrow WRITE_B(victime \leftarrow B) \rightarrow READ_B((drapeau[A] \wedge victime = B) = \perp) \rightarrow SC_B$
Supposons qu'Alice a écrit en dernier dans la variable victime, alors

$$WRITE_B(victime \leftarrow B) \rightarrow WRITE_A(victime \leftarrow A)$$

Or Alice a lu $READ_A((drapeau[B] \wedge victime = A) = \perp)$. Donc

$$WRITE_B(drapeau[B] \leftarrow \top) \rightarrow READ_A(drapeau[B] = \perp)$$

. Or la seule écriture possible dans drapeau[B] est dans le LOCK pour Bob (car Bob est en section critique). drapeau[B] change de valeur sans aucune écriture, ce qui est absurde (sauf si un neutrino est passé par là). Donc Alice et Bob ne sont pas simultanément en section critique.

Pour montrer l'absence de famine, montrons qu'aucun thread n'est bloqué (i.e que chaque thread qui fait un LOCK finit par arriver en section critique). On suppose que Bob a exécuté son LOCK et Alice est en section critique, puis exécute son UNLOCK. Donc $CS_A \rightarrow WRITE_A(drapeau[X] \leftarrow \perp)$ (1). Si Alice exécute un nouveau LOCK juste après : $WRITE_A(drapeau[X] \leftarrow \top) \rightarrow WRITE_A(victime \leftarrow A)$ (2). Bob était bloqué sur "Tant que drapeau[A] et victime = B" :

- Si cette condition est évaluée juste après (1), elle est fausse car drapeau[A] = \perp .
- Si elle est évaluée après (2) elle est fausse car victime = A.

Dans les deux cas, c'est faux donc Bob entre en SC_B . Ceci garantit d'ailleurs l'alternance entre SC_A et SC_B , donc l'équité. Conséquence : il n'y a pas de blocage. ■

Remarque. En pratique, on n'utilise pas l'algorithme de PETERSON. En effet :

- L'algorithme a une attente active (très inefficace)
- La correction suppose que les instructions d'un même thread sont bien exécutées dans l'ordre, ce qui est faux sur la plupart des processeurs, qui réarrangent comme ils veulent les instructions tant que le résultat final est le même.

De plus, l'algorithme de PETERSON ne fonctionne qu'avec deux threads. On s'interroge alors sur la manière de faire des mutex avec plus de processus.

3.2 Mutex à n processus

3.2.1 Généralisation de Peterson (HP)

Une généralisation de l'algorithme de PETERSON aurait comme objectif de ne laisser, parmi n processus, qu'un seul entrer en section critique. On remarque que l'algorithme de PETERSON, plus que de choisir un processus y entrant, en élimine un qui se sacrifie en se désignant comme victime. On élabore donc une généralisation en n étapes où, à chaque étape, on élimine un processus parmi ceux restants. Un seul parvient alors au sommet de la pyramide.

Algorithme 3.2 Algorithme de PETERSON généralisé

Entrée : n processus

```
fonction LOCK( $i$ ) :  
  Pour  $j$  allant de 1 à  $n - 1$ , faire :  
    niveau[ $i$ ]  $\leftarrow j$   
    victime[ $j$ ]  $\leftarrow i$   
    Tant que  $\exists k \neq i$  tq niveau[ $k$ ]  $\geq j$  et victime[ $j$ ] =  $i$ , faire :  
      ATTENDRE  
    Fin tant que  
  Fin pour  
Fin fonction  
  
fonction UNLOCK( $i$ ) :  
  niveau[ $i$ ]  $\leftarrow 0$   
Fin fonction
```

3.2.2 Boulangerie de Lamport

Algorithme 3.3 Boulangerie de LAMPORT

```
fonction LOCK( $i$ ) :  
  drapeau[ $i$ ]  $\leftarrow \top$   
  ticket[ $i$ ]  $\leftarrow 1 + \max_j(\text{ticket}[j])$   
  Tant que  $\exists j$  tel que drapeau[ $j$ ] et (ticket[ $j$ ],  $j$ ) < (ticket[ $i$ ],  $i$ ), faire :  
    ATTENDRE  
  Fin tant que  
Fin fonction  
  
fonction UNLOCK( $i$ ) :  
  drapeau[ $i$ ]  $\leftarrow \perp$   
Fin fonction
```

Proposition 3.6. *La boulangerie de LAMPORT garantit l'exclusion mutuelle.*

Démonstration. Les tickets de chaque thread sont croissants avec le temps. Donc, si deux processus distincts i et j sont en section critique, alors ils ont lu :

$$\text{read}_i((\text{ticket}[j], j) \geq (\text{ticket}[i], i)) \quad \text{et} \quad \text{read}_i((\text{ticket}[i], i) \geq (\text{ticket}[j], j)).$$

Donc $\text{ticket}[i] = \text{ticket}[j]$ puis $i = j$, ce qui est absurde. ■

Proposition 3.7. *L'algorithme de la boulangerie est une file (ordonnée par numéro de processus quand les tickets sont égaux).*

Donc il n'y a pas de blocage ou de famine, et il y a équité (c'est à dire que chaque thread entre en section critique aussi souvent que les autres).

De même que l'algorithme de PETERSON, la boulangerie de LAMPORT est un algorithme purement théorique qui suppose que l'ordre d'exécution est bien celui-ci. Pour illustrer ce point :

Exemple. Si on inverse les instructions de drapeau[i] ticket [i] :

```
Thread nř1, thread nř42 :
READ1(max(tickets) = 17)
READ42(max(tickets) = 17)
WRITE42(tickets[42] = 17)
WRITE42(drapeau[42] =  $\top$ )
READ42(drapeau[42] =  $\top$ )
SC42
Mais parallèlement :
READ42(drapeau[1] =  $\perp$ )
WRITE1(tickets[1] = 18)
WRITE1(drapeau[1] =  $\top$ )
READ1((tickets[1], 1) < (tickets[42], 42))
SC1
```

De plus, il est en pratique impossible de recourir à un ensemble fini pour indiquer les tickets. Par conséquent la menace des *overflows* (dépassement de capacité) doit être prise en compte.

3.3 Sémaphores

Définition 3.8 (Sémaphore). C'est une structure de données qui stocke un entier n et possède deux opérations : WAIT et SIGNAL.

L'opération WAIT :

- si $n > 0$, WAIT décrémente n ,
- sinon, WAIT bloque tant que $n \leq 0$ (en fait, $n = 0$) puis décrémente n .

L'opération SIGNAL incrémente n , et débloque n threads bloqués par un WAIT

Remarque. Un sémaphore permet d'implémenter un mutex :

$s \leftarrow \text{semaphore}(1)$

fonction LOCK :

WAIT

Fin fonction

fonction UNLOCK :

SIGNAL

Fin fonction

Remarque. On ne sait pas si un sémaphore garantit l'absence de famine ou l'équité (la description de WAIT ne le précise pas. En général, on suppose qu'un sémaphore a ces propriétés.

À Méditer : Si on a uniquement des sémaphores non équitables, peut on créer des sémaphores équitables ? (oui)

3.4 Implémentations

Cette section présente les implémentations des sémaphores et des processus légers dans les deux langages au programme. Celles-ci ne sont pas à connaître et seront redonnées en cas de besoin.

3.4.1 En C

```
#include <pthread.h>

void* f(void* value) {
    // ...
}

pthread_t thread;
pthread_create(&thread, NULL, f, &value);
pthread_join(thread, NULL);
```

Listing 1 : Création de processus légers en C

```
#include <pthread.h>

// On peut initialiser directement à la création d'une variable.
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// Sinon, il existe une fonction pour le faire plus tard.
pthread_mutex_initialize(&mutex);
// Verrouillage
pthread_mutex_lock(&mutex);
// ... section critique ...
// Déverrouillage
pthread_mutex_unlock(&mutex);
```

Listing 2 : Création d'exclusions mutuelles en C

```
#include <semaphore.h>

sem_t semaphore;
sem_init(&semaphore, 0, capacite_initiale);
sem_wait(&semaphore); // Décrémenter le sémaphore, bloquant s'il est arrivé à zéro
sem_post(&semaphore); // Incrémenter le sémaphore, éventuellement réveille des threads
↪ bloqués
```

Listing 3 : Création de sémaphores en C

3.4.2 En OCaml

En OCaml, les processus légers ne tournent pas en parallèle mais à tour de rôle. Ils doivent volontairement s'arrêter de temps en temps si on veut que les autres fils s'exécutent de manière concurrente. Un grand nombre de fonctions de la bibliothèque standard contient de manière cachée un appel pour suspendre le fil actuel.

```

(* Le module Thread contient un type abstrait qui représente un thread *)
type Thread.t
(* Création à partir d'une fonction et du paramètre à lui donner.
   Le thread tourne immédiatement.
   Il n'y a aucun moyen de récupérer la valeur renvoyée. *)
val Thread.create : ('a -> 'b) -> 'a -> Thread.t
(* Attendre la fin d'un processus léger *)
val Thread.join : Thread.t -> unit
(* Tenter de se mettre en sommeil et passer la main explicitement *)
val Thread.yield : unit -> unit

(* Exemple d'usage *)
let f x = (* *)
let thread = Thread.create f value
let () = Thread.join thread

```

Listing 4 : Création de processus légers en OCaml

```

type Mutex.t
val Mutex.create : unit -> Mutex.t
val Mutex.lock : Mutex.t -> unit
val Mutex.unlock : Mutex.t -> unit

let mutex = Mutex.create ()
let () = Mutex.lock mutex
(* *)
let () = Mutex.unlock mutex

```

Listing 5 : Création d'exclusions mutuelles en OCaml

```

type Semaphore.Counting.t
val Semaphore.Counting.make : int -> Semaphore.Counting.t
val Semaphore.Counting.acquire : Semaphore.Counting.t -> unit
val Semaphore.Counting.release : Semaphore.Counting.t -> unit

let sem = Semaphore.Counting.make capacite
let () = Semaphore.Counting.acquire sem
let () = Semaphore.Counting.release sem

```

Listing 6 : Création de sémaphores en OCaml

3.5 Quelques exemples au programme

3.5.1 Dîner de philosophes

L'algorithme qui suit est écrit avec un peu plus de légèreté, les instructions sont pleinement en français. Imaginons un ensemble de philosophes dînant ensemble autour d'une table ronde, chacun souhaite déguster avec soin son plat, les philosophes étant des personnes fortes bien civilisées mangent avec deux fourchettes respectivement partagées avec leurs voisins. Par conséquent s'ils désirent se rassasier, ils doivent attendre que leurs voisins arrête de se nourrir. Le danger, qu'ils restent tous bloqués en cas de mauvaises instructions².

2. d'où le terme *famine* pour désigner une situation où les fils d'exécution restent bloqués

Algorithme 3.4 Diner des philosophes

3.5.2 Rendez-vous

Voir exercice 3.2

3.5.3 Producteurs-consommateurs

Le but de cet algorithme est d'établir un système dans lequel d'un côté certaines attendent tour à tour une ressource, mais que d'un autre côté celle-ci est commandée par un tiers. Dans les écritures qui suivent, l'instruction PUSH et POP correspondent à celles effectuées sur une file qui représentent un stock de produits.

Algorithme 3.5 Producteurs consommateurs

mutex \leftarrow semaphore(1)

stock \leftarrow semaphore(0)

fonction PRODUCTEURS :

 WAIT(mutex)

 PUSH

 SIGNAL(stock)

 SIGNAL(mutex)

Fin fonction

fonction CONSOMMATEURS :

 WAIT(stock)

 WAIT(mutex)

 POP

 SIGNAL(mutex)

Fin fonction

3.5.4 Barrière de synchronisation

On implémente ici une barrière de synchronisation entre plusieurs fils d'exécution. Autrement dit, d'une fonction garantissant que les fils d'exécution s'attendent avant de continuer une autre tâche. Elle sera par ailleurs réutilisable.

Exercices

Puzzles classiques

Exercice 3.1. Montrer qu'il est possible d'implémenter un mutex avec un sémaphore.

Exercice 3.2. Alice doit exécuter une tâche A_1 et Bob une tâche B_1 . La tâche de Bob doit impérativement être exécutée après la fin de celle d'Alice. Proposer un code pour chaque processus qui accomplit ceci.

(Rendez-vous.) Alice a désormais deux tâches A_1 et B_1 , et Bob a B_1 et B_2 . Les tâches d'indice 1 doivent s'exécuter avant celles d'indice 2. Écrire les processus correspondants.

Exercice 3.3. Dans une soirée APH, seulement n personnes sont autorisées à danser simultanément, les autres doivent rester assis. Implémenter le code de chaque danseur.

Exercice 3.4. Une barrière est un point de rendez-vous que les n processus doivent passer simultanément. Autrement dit, il s'agit d'une fonction *barriere* qui est bloquante tant que tous les n processus ne l'ont pas appelée, puis qui libère tous les processus. Implémenter une barrière (sans attente active). La première version n'a pas besoin d'être réutilisable.

Algorithme 3.6 Barrière de synchronisation

Entrée : le nombre de fils n

```
 $k \leftarrow 0$                                  $\triangleright$  c'est le nombre de fils passés
 $s \leftarrow \text{semaphore}(0)$ 
 $s' \leftarrow \text{semaphore}(0)$ 
fonction ENTRER :
    LOCK
     $k \leftarrow k + 1$ 
    Si  $k = n$  alors
        Pour  $i$  de 1 à  $n$ , faire :
            SIGNAL( $s$ )
        Fin pour
    Fin si
    UNLOCK
    WAIT( $s$ )
Fin fonction

fonction SORTIR :
    LOCK
     $k \leftarrow k - 1$ 
    Si  $k = 0$  alors
        Pour  $i$  de 1 à  $n$ , faire :
            SIGNAL( $s'$ )
        Fin pour
    Fin si
    UNLOCK
    WAIT( $s$ )
Fin fonction
```

Un *tourniquet* permet de laisser passer un nombre fixé n de processus en section critique, puis est bloquant jusqu'à ce que certains processus sortent de la section critique. Il garantit qu'il y a toujours au plus n processus en section critique. Implémenter un tourniquet.

Implémenter une barrière réutilisable.

Exercice 3.5. Pour danser, il faut deux danseurs : un danseur à talons plats et un danseur à talons hauts. Quand un danseur, quel que soit son type, arrive dans la salle, s'il trouve un danseur de l'autre type, ils dansent ensemble. Sinon il attend l'arrivée d'un tel danseur. Implémenter la ginguette.

Votre solution s'assure-t-elle que chaque danseur danse bien avec celui avec lequel il est entré dans la salle ? Si ce n'est pas le cas, la modifier pour assurer cette propriété.

Exercice 3.6. Le travail est partagé entre plusieurs processus légers. Certains sont des *producteurs*, d'autres des *consommateurs*. Les producteurs créent des objets, les consommateurs les détruisent. Il y a un peu de stock d'objets, que les producteurs peuvent remplir tant qu'il n'est pas plein. Les consommateurs se servent en objets dans le stock. L'accès au stock n'est possible que par un processus à la fois.

Quand un consommateur veut se servir dans un stock vide, il doit être mis en attente. Il en est de même pour un producteur qui veut ajouter un élément quand le stock est plein.

1. Écrire le code qui permet aux producteurs et aux consommateurs de respecter ces règles de synchronisation quand la capacité du stock est infinie.
2. Écrire le code correspondant quand la capacité du stock est finie.

Entrelacement

On considère ici les fonctions `f1` et `f2`. On va lancer un thread pour chacune de ces deux fonctions.

```
int x = 0;
int y = 0;
void* f1(void* arg)
{
    if(x == 0) {
        x = x + 1;
        y = y + 1;
    }
}
void* f2(void* arg)
{
    if(x == 0) {
        x = x + 1;
        y = y + 2;
    }
}
```

Question 3.1. En supposant que les opérations $x = x + 1$, $y = y + 1$ et $y = y + 2$ sont atomiques (c'est-à-dire que, pour chaque ligne, la lecture de la variable, l'addition puis l'affectation sont réalisées en une seule opération élémentaire) donner les valeurs finales possibles pour x et y .

Lecteurs, rédacteurs

On souhaite gérer un système de concurrence lecteurs/rédacteurs où plusieurs lecteurs et plusieurs rédacteurs agissent en parallèle sur un document. Les lecteurs se contentent de lire le document et les rédacteurs le modifient. Les lecteurs peuvent donc être plusieurs à lire simultanément sans que cela n'implique de problème de concurrence à condition qu'aucun rédacteur ne soit en train de modifier le document. Un seul rédacteur à la fois peut travailler sur le document et si un rédacteur est en train de travailler sur le document alors aucun lecteur ne peut plus y accéder. On propose l'utilisation de la structure C suivante où on dispose d'un sémaphore binaire (c'est-à-dire un sémaphore qui ne peut prendre que les valeurs 0 et 1, il est initialisé à 1). La variable `nreaders` est partagée et compte le nombre de lecteurs en train de consulter le document.


```

struct rwlock_s {
    sem_t writelock;
    int nbreaders;
}
typedef struct rwlock_s rwlock_t;

void rwlock_init(rwlock_t* l)
{
    sem_init(&(l->writelock), 0, 1);
    nbreaders = 0;
}

```

Nous allons écrire une fonction d'accès (`rwlock_acquire_readlock`) et une fonction de sortie de la section de lecture (`rwlock_release_readlock`) ainsi qu'une fonction d'accès (`rwlock_acquire_writelock`) et une fonction de sortie (`rwlock_release_writelock`) d'une section d'écriture qui vont respecter les contraintes. Voici une première proposition :

```

void rwlock_acquire_readlock(rwlock_t *lock)
{
    lock->nbreaders++;
    if(lock->nbreaders == 1) {
        sem_wait(&lock->writelock);
    }
}

void rwlock_release_readlock(rwlock_t *lock)
{
    lock->nbreaders--;
    if(lock->nbreaders == 0) {
        sem_post(&lock->writelock);
    }
}

void rwlock_acquire_writelock(rwlock_t *lock)
{
    sem_wait(&lock->writelock);
}

void rwlock_release_writelock(rwlock_t *lock)
{
    sem_post(&lock->writelock);
}

```

Question 3.2. Une erreur s'est glissée dans la fonction `rwlock_init`. L'identifier et la corriger.

Question 3.3. Aurait-on pu utiliser un autre type de variable à la place du sémaphore `write_lock`?

Question 3.4. Expliquer les rôles des `if` dans les fonctions `rwlock_acquire_readlock` et `rwlock_release_readlock`.

Question 3.5. Pourquoi deux rédacteurs ne pourront pas écrire simultanément ?

Question 3.6. Pourquoi les agents (lecteurs et rédacteurs) utilisent-ils tous le même sémaphore ?

Question 3.7. Quel problème de concurrence apparaît dans le code proposé ? Expliquer comment modifier le code pour résoudre ce problème.

On considère maintenant les fonctions `reader` et `writer` suivantes.

```

int read_loops;
int write_loops;
int counter = 0;
rwlock_t mutex;

void* reader(void *arg)
{
    int i;
    int local = 0;
    for (i = 0; i < read_loops; i++) {
        rwlock_acquire_readlock(&mutex);
        local = counter;
        rwlock_release_readlock(&mutex);
        printf("read %d\n", local);
    }
    printf("read done\n");
    return NULL;
}

void* writer(void *arg)
{
    int i;
    for (i = 0; i < write_loops; i++) {
        rwlock_acquire_writelock(&mutex);
        counter++;
        rwlock_release_writelock(&mutex);
    }
    printf("write done\n");
    return NULL;
}

```

Question 3.8. Que représente la variable counter ?

Question 3.9. Recopier et compléter le main dont le rôle est de lancer les fonctions précédentes :

```

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "usage: rwlock readloops writeloops\n");
        exit(1);
    }
    read_loops = atoi(argv[1]);
    write_loops = atoi(argv[2]);

    rwlock_init(&mutex);

    pthread_create(
        , NULL,
        , NULL);
    pthread_create(
        , NULL,
        , NULL);

    return 0;
}

```

Question 3.10. Supposons que read_loops=5 et write_loops=3. Parmi les affichages suivants, dire, en justifiant, lesquels correspondent à une exécution possible :

(a) read 0	(c) read 3	(e) read 1	(g) read 0
read 3	read 3	read 3	read 2
read 3	write done	read 2	write done
read 3	read 3	read 3	read 3
read 3	read 3	read 3	read 3
read 3	read 3	read done	read 3
read done	read done	write done	read done
write done			
(b) read 3	(d) read 0	(f) read 0	(h) read 0
read 3	read 1	read 0	read 2
read 3	read 2	read 0	write done
read 3	read 2	read 0	read 2
read 3	read 3	read 0	read 3
read done	read done	read done	read 3
write done	write done	write done	read done

Le coiffeur

La boutique du coiffeur est composée d'une salle d'attente contenant `NB_CHAISES` chaises et du salon où se trouve la chaise du coiffeur. Lorsque le coiffeur a fini de couper les cheveux d'un client, il fait entrer le client suivant dans le salon. Si la salle d'attente est vide, le coiffeur s'y installe pour dormir. Si un client trouve le coiffeur endormi, il le réveille. Sinon, il s'installe dans la salle d'attente s'il reste de la place (et rentre chez lui sinon).

Dans cet exercice vous utiliserez des sémaphores dont vous préciserez les valeurs initiales et vous pourrez vous contenter de noter `wait(s)` et `post(s)` pour les opérations d'acquisition et de restitution d'un sémaphore noté `s`.

Algorithme 3.7 Protocole du coiffeur

Initialisation : `places = NB_CHAISES` // représente le nombre de places disponibles

Tant que True, **faire** :

Si `places < NB_CHAISES` **alors**

 Incrémenter `places`

 Couper les cheveux

Fin si

Fin tant que

Algorithme 3.8 Protocole de chaque client

Si `places > 0` **alors**

 Décrémenter `places`

 Se faire couper les cheveux

Sinon

 Rentrer chez soi avec les cheveux longs

Fin si

Question 3.11. Il s'agit maintenant d'ajouter les synchronisations nécessaires au programme ci-dessus. Le premier problème à résoudre est une condition de compétition entre les clients lorsqu'ils rentrent dans la salle d'attente. Corrigez ce problème.

Question 3.12. Assurez-vous ensuite que le coiffeur ne commence pas à couper les cheveux tant que le client n'est pas prêt.

Question 3.13. Enfin, assurez-vous que le client ne s'assoit pas sur le siège tant que le coiffeur n'est pas prêt.

Chapitre 4

Classes de complexité

Sommaire

4.0	Banalités autour de SAT (HP)	61
4.0.1	Sudoku	61
4.0.2	Exécuter un programme	62
4.1	Problèmes de décision	62
4.2	Réduction de problèmes	64
4.2.1	Exemples en logique (HP)	65
4.2.2	Exemples dans les graphes (HP)	67
4.3	Autres classes de complexité (HP)	69
4.4	Décidabilité	70
4.4.1	Histoire sympa	70
4.4.2	Définition	71
	Exercices	72

C'est la jungle

F.Hatat

4.0 Banalités autour de SAT (HP)

4.0.1 Sudoku

Avec suffisamment de variables, on peut représenter la règle du jeu du Sudoku et la validité d'une grille par une formule de la logique propositionnelle satisfiable.

Pour cela, on associe à chacune des 81 cases (i, j) de la grille 9 variables $x_{i,j}^k$, vraies si et seulement si le chiffre k est placé en case (i, j) .

Le fait que la première ligne ne contienne qu'une seule fois le chiffre 4 s'exprime par le fait que la formule suivante est satisfiable :

$$\bigwedge_{j=0}^8 \bigwedge_{\substack{j'=0 \\ j' \neq j}}^8 \neg x_{0,j}^4 \wedge \neg x_{0,j'}^4$$

On peut faire de même pour toutes les lignes, les colonnes et les petits carrés de côté 3, ce qui donne bien une formule de la logique propositionnelle.

Par conséquent, si on sait décider si une formule est satisfiable, alors on sait décider si une grille de Sudoku possède une solution. Pour savoir quelle taille fait la formule à construire et avoir les détails, on peut regarder le sujet CentraleSupélec 2014 option informatique. On peut montrer que la formule est de taille polynomiale par rapport à la taille de la grille (si on s'autorise une variante du Sudoku de taille quelconque).

4.0.2 Exécuter un programme

On peut imaginer un ordinateur comme un objet agissant sur des cases de mémoire, indicées par \mathbb{N} qui contiennent chacune un seul bit valant 0 ou 1. À chacune de ces cases, on associe des variables x_k^t qui sont vraies si et seulement si la case k vaut un au temps t . Un algorithme \mathcal{A} est alors une suite d'instructions qui changent l'état de cases en mémoire, elles sont donc de la forme :

$$\bigwedge_{k \in I} x_k^t \implies \bigwedge_{k' \in I'} x_{k'}^{t+1}.$$

On résume ainsi l'idée que l'exécution d'un programme revient à l'évaluation d'une grosse conjonction.

On suppose que la réponse attendue est un seul bit à écrire en 0. \mathcal{A} répond **vrai** sur une entrée donnée si et seulement si la formule

$$\underbrace{(x_i^0 \wedge \neg x_n^0 \wedge \dots \wedge x_n^0)}_{\text{variables d'entrées}} \wedge \underbrace{\mathcal{P}}_{\text{instructions}} \wedge \underbrace{x_0^{t_{\text{final}}}}_{\text{résultat}},$$

est satisfiable (ce fait porte le nom de Théorème de Cook). Par conséquent, on aperçoit que le problème de satisfiabilité d'une formule logique est un problème de tout premier ordre, ce qui mène à vouloir classifier les différents problèmes qui existent. Pour davantage d'informations sur la conception théorique de la machine, voir l'annexe hors programme sur les machines de Turing.

4.1 Problèmes de décision

On rappelle la définition $\mathbb{B} := \{\top, \perp\}$ de l'ensemble des booléens.

Définition 4.1. Un problème de décision est une fonction $f : E \rightarrow \mathbb{B}$.

Remarque. La plupart du temps, ces derniers se présentent sous la forme d'une question concernant une propriété que devrait vérifier un certain objet.

Exemples. — SAT est un problème de décision : $\text{SAT}(f)$ répond vrai si et seulement si la formule f est satisfiable.

- Un graphe est-il connexe ?
- Un entier est-il premier ? Ce qui correspond à $\text{PRIME}(\cdot) : \mathbb{N} \rightarrow \mathbb{B}$
- Soit $k \in \mathbb{N}$. Un graphe est-il k -coloriable ?

Exemple (Contre exemple d'un problème qui n'est pas un problème de décision). Soit G un graphe non orienté. La question "avec quel nombre minimum de couleurs peut-on colorier G ?" ne permet pas d'obtenir un problème de décision puisque le résultat attendu est un entier et non un booléen.

Définition 4.2 (Problème d'optimisation). Soient E et S deux ensembles (respectivement d'entrées et de solutions) munis d'une relation $\mathcal{R} \subset E \times S$ et une fonction dit de coût $c : S \rightarrow \mathbb{R}$.

Un problème d'optimisation est une fonction $f : E \rightarrow S$ telle que

$$\forall e \in E, \quad c(f(e)) = \min\{c(s)\}_{s \in \bar{e}},$$

où $\bar{e} := \{s \in S : s\mathcal{R}e\}$ est la classe de relation d'une entrée e .

Remarque (Problème de décision avec seuil). Si on a un problème d'optimisation, on peut le changer en problème de décision dont l'entrée est un couple $(k, e) \in \mathbb{R} \times E$ et répondant à la question "existe-t-il une solution de coût plus petit que k ".

Avec de bonnes hypothèses, le problème de décision avec seuil permet de résoudre le problème d'optimisation en $\log(n)$ étapes où n est le nombre de valeurs possibles pour k .

Définition 4.3. Un algorithme \mathcal{A} est un programme en C ou en OCaml avec mémoire illimitée.

Définition 4.4 (Complexité d'un algorithme). C'est le temps de calcul, en nombre d'opérations élémentaires. C'est une fonction qui s'exprime en fonction de la taille de l'entrée.

Exemples. — L'entrée de PRIME est un entier naturel k avec $|k| = O(\log(k))$ si on l'écrit dans une base plus grande que 2.

— Un graphe à n sommets non orienté muni de sa matrice d'adjacence peut s'écrire comme une entrée de taille $\log(n) + n^2$.

Définition 4.5. Soit un algorithme \mathcal{A} de complexité C . \mathcal{A} est dit polynomial lorsqu'il existe $P \in \mathbb{R}[X]$ tel que $C(n) = O(P(n))$.

Définition 4.6. Un problème de décision f est polynomial lorsqu'il existe un algorithme polynomial \mathcal{A} tel que

$$\forall e \in E, \mathcal{A}(e) = \text{vrai} \iff f(e) = \top.$$

Définition 4.7 (P). On désigne par **P** l'ensemble des problèmes de décision dont la complexité est polynomial.

Exemples. — MIN(k, L) répondant à la question " k est-il le minimum de la liste L ?" alors MIN \in **P**.

— PRIME \in **P** (non évident et c'est un résultat très récent datant du 6 août 2002).

— Soient G un graphe non orienté, k un entier, et **tab** un tableau qui à chaque sommet de G associe une couleur. Considérons VERIF_K_COLOR(G, tab) répondant à la question "**tab** est-il une k -coloration valide de G ?" alors VERIF_K_COLOR \in **P**.

Définition 4.8 (Certificat). Soit $f : E \rightarrow \mathbb{B}$ un problème de décision. Soit \mathcal{C} un ensemble dont les éléments sont appelés certificats. On dit que $g : E \times \mathcal{C} \rightarrow \mathbb{B}$ est un problème de vérification de f lorsque

$$\forall e \in E, f(e) = \top \iff \exists c \in \mathcal{C}, g(e, c) = \top.$$

Remarque. Informellement, on ne demande un certificat que dans les cas où $f(e) = \top$. Si $f(e) = \perp$, on demande qu'aucun certificat ne convienne, c'est la traduction de l'équivalence qui tient de définition.

Définition 4.9. On garde les notations de la définition précédente. Un certificat $c \in \mathcal{C}$ est dit polynomial lorsqu'il existe $P \in \mathbb{R}[X]$ tel que $|c| = O(P(|e|))$.

Exemple. On définit pour tout entier k , COPRIME(k) := \neg PRIME(k). Autrement dit COPRIME(k) renvoie vrai si et seulement si k n'est pas premier. Un certificat possible pour COPRIME est un couple d'entiers (a, b) tel que $(a \neq 1) \wedge (b \neq 1)$ et $ab = k$. Ainsi g vérifie :

- $a \neq 1$
- $b \neq 1$
- $ab = k$

Exemple (Certificat pour PRIME?). Liste de tous les diviseurs jusqu'à \sqrt{k} ?

- Ça n'aide pas du tout : le problème n'est pas plus simple à résoudre avec ce certificat.
- Ce certificat n'est pas de taille polynomiale en k (la taille d'un entier, c'est son nombre de chiffres, donc en $\log k$).

On sait que PRIME \in **NP** mais la preuve n'est pas du tout évidente. Cela vient du fait que PRIME \in **P**.

Définition 4.10 (NP). On désigne par **NP** l'ensemble des problèmes de décision dont il existe un problème de vérification $g : E \times \{0, 1\}^* \rightarrow \mathbb{B}$ pour celui-ci polynomial en temps et un polynôme $P \in \mathbb{R}[X]$ tel que :

$$\forall e \in E, f(e) = \top \iff \exists c \in \{0, 1\}^*, |c| = O(P(|e|)) \wedge g(e, c) = \top$$

Exemples. — COPRIME \in **NP**.

- PRIME \in **NP** ce qui n'est pas du tout évident.
- Soit G un graphe non orienté. K_COLOR(G) répondant à la question " G est-il k -colorable?". K_COLOR \in **NP** dont un certificat est une coloration et dont le problème de vérification est donné grâce à VERIF_K_COLOR qui est polynomial.

Théorème 4.11. On dispose de l'inclusion **P** \subseteq **NP**.

Démonstration. Soit $f : E \rightarrow \mathbb{B}$ un problème de **P**. Posons $C := \{()\}$ puis pour tout $(e, c) \in E \times C$, $g(e, c) := f(e)$ de sorte que pour tout $c \in C$, $c = ()$ si bien que $|c| = O(1)$ ce qui est polynomial en $|e|$. Par conséquent $g \in \mathbf{P}$ car $f \in \mathbf{P}$ donc $f \in \mathbf{NP}$ ce qu'on voulait montrer. ■

Théorème 4.12. $SAT \in \mathbf{NP}$

Démonstration. car soit f une formule propositionnelle. On se sert d'un témoin de satisfiabilité ρ de f pour construire un certificat pour SAT. On note x_1, \dots, x_n les différentes variables de f . Dans premier temps, on cherche à encoder les valuations. Pour tout $k \in \llbracket 1, n \rrbracket$

$$\llbracket \rho \rrbracket_k := \begin{cases} 1 & \text{si } k \leq n \wedge \rho(x_k) = \top, \\ 0 & \text{sinon} \end{cases},$$

puis posons

$$\llbracket \rho \rrbracket := \prod_{k=1}^n \llbracket \rho \rrbracket_k \in \{0, 1\}^*.$$

Soit $c \in \{0, 1\}^*$. On construit ensuite un problème de vérification pour SAT en :

- Comménçant par vérifier que $|c| = n$.
- Puis en évaluant f avec la valuation c et renvoie cette valeur.

On note g la fonction ainsi construite et remarquons que g s'exécute en $O(|f| + |c|)$. Or, $|c| \leq |f|$. (et on peut rejeter les c invalides en $n + 1$ étapes). Donc g s'exécute en temps $O(|f|)$.

- Ainsi, dans l'hypothèse où f est satisfiable, c'est à dire si $SAT(f) = \top$, alors il existe une valuation ρ telle que $\rho(f) = \top$ et alors puisque $g(f, \llbracket \rho \rrbracket) = \rho(f)$, alors $g(f, \llbracket \rho \rrbracket) = \top$
- Réciproquement s'il existe c tel que $g(f, c) = \top$. Alors $|c|$ est le nombre de variables de f par définition de g et on peut donc définir

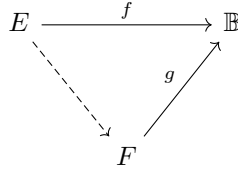
$$\rho(x_k) := \begin{cases} \top & \text{si } c_k = 1 \\ \perp & \text{si } c_k = 0 \end{cases},$$

de sorte que $g(f, \llbracket \rho \rrbracket)$ calcule $\rho(f)$ or $g(f, c) = \top$ par hypothèse donc $\rho(f) = \top$ si bien f est satisfiable et donc $SAT(f) = \top$.

Par définition $SAT \in \mathbf{NP}$ ce qui achève la preuve. ■

4.2 Réduction de problèmes

Définition 4.13 (Réduction). On considère deux problèmes de décision : $f : E \rightarrow \mathbb{B}, g : F \rightarrow \mathbb{B}$. Une réduction de f à g c'est une fonction $\varphi : E \rightarrow F$ telle que $g(\varphi(e)) = \top \iff f(e) = \top$. Autrement dit, φ fait commuter le triangle



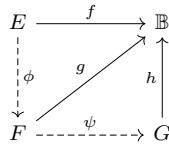
On dit que la réduction φ est polynomiale lorsqu'il existe un algorithme qui calcule $\varphi(e)$ en temps polynomial en $|e|$.

Corollaire 4.14. Il faut un $O(1)$ pour écrire chaque bit de la sortie $\varphi(e)$ donc $|\varphi(e)|$ par un polynôme en $|e|$. S'il existe une réduction polynomiale de f à g on pourra noter

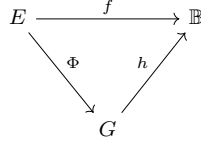
$$f \leq_P g$$

Lemme 4.15. La relation \leq_P est transitive.

Démonstration. En effet, soit $f : E \rightarrow \mathbb{B}$, $g : F \rightarrow \mathbb{B}$, $h : G \rightarrow \mathbb{B}$ trois problèmes de décision tels que $f \leq_P g$ et $g \leq_P h$. On a ainsi le carré à diagonal commutatif suivant



La composition $\Phi := \psi \circ \phi$ rend alors commutatif le triangle :



si bien que Φ est une réduction de f à h et puisque la composée de deux polynômes est un polynôme, la réduction Φ est polynomiale. On a ainsi par définition

$$f \leq_P h,$$

ce qu'on voulait montrer. ■

Lemme 4.16. *Soit $f : E \rightarrow \mathbb{B}$ dans **NP** et $F \subseteq E$. Le sous-problème $\tilde{f} := f|_F$ est dans **NP**.*

Démonstration. Par hypothèse, il existe $g : E \times \{0, 1\}^* \rightarrow \mathbb{B}$ et $P \in \mathbb{R}[X]$ tel que

$$\forall e \in E, \quad f(e) = \top \iff \exists c \in \{0, 1\}^*, \quad |c| = O(P(|e|)) \wedge g(e, c) = \top.$$

Le problème de vérification $\tilde{g} := g|_{F \times \{0, 1\}^*}$ hérite donc des propriétés de g , en particulier

$$\begin{aligned}
\forall e \in F, \quad \tilde{f} = f|_F(e) = f(e) = \top &\iff \exists c \in \{0, 1\}^*, \quad |c| = O(P(|e|)) \wedge g(e, c) = \top \\
&\iff \exists c \in \{0, 1\}^*, \quad |c| = O(P(|e|)) \wedge \\
&\quad \top = g(e, c) = g|_{F \times \{0, 1\}^*}(e, c) = \tilde{g}(e, c),
\end{aligned}$$

d'où le résultat par définition. ■

Définition 4.17. Un problème f est **NP-complet** lorsque :

- (i) f est **NP**
- (ii) $\forall g \in \mathbf{NP}, g \leq_P f$

Théorème 4.18 (Théorème de réduction). *Soit f un problème **NP-complet**. Soit g un problème de décision de **NP** et l'on suppose que $f \leq_P g$. Alors g est **NP-complet**.*

Démonstration. Dans les conditions de l'énoncé du Théorème, on a

$$(\forall h \in \mathbf{NP}, \quad h \leq_P f) \wedge (f \leq_P g),$$

donc d'après le lemme 4.15

$$\forall h \in \mathbf{NP}, \quad h \leq_P g,$$

autrement dit, g est **NP-complet**. ■

Remarque. Si on sait résoudre un problème **NP-complet** en temps polynomial alors on sait résoudre tout problème **NP** en temps polynomial.

Théorème 4.19 (de Cook, admis). *SAT est **NP** complet.*

Remarque. Pour montrer qu'un problème f est **NP** complet, il suffit de prouver, en vertu du Théorème 4.18 que :

- f est **NP**
- $\text{SAT} \leq_P f$

Précisons le deuxième point. Cela revient à :

- Prendre une instance I de SAT quelconque
- Transformer I en $\varphi(I)$ en temps polynomial, où $\varphi(I)$ est une entrée de f de telle sorte que $\text{SAT}(I) = \top \iff f(\varphi(I)) = \top$.

4.2.1 Exemples en logique (HP)

Dans cette section, nous allons montrer la **NP-complétude** de certains problèmes dérivés de SAT.

Définition 4.20. Une formule 3-SAT est une formule sous forme normale conjonctive telle que chaque clause est de taille au plus 3 littéraux.

Exemple. Typiquement

$$(a \vee b \vee \bar{a}) \wedge (a \vee c \vee d) \wedge (a \vee \bar{a}) \wedge (z)$$

est une formule de 3SAT.

Dans ce qui suit, on cherche à réduire les formules de taille quelconque en des formules 3-SAT et à cet effet, on développe la transformée de Tseitin. Mais avant toutes choses, quelques méditations logiques :

Lemme 4.21. *On dispose des équivalences suivantes*

- $x \longleftrightarrow (y \wedge z) \equiv (\neg x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z \vee x)$
- $x \longleftrightarrow (y \vee z) \equiv (\neg x \vee y \vee z) \wedge (\neg y \vee x) \wedge (\neg z \vee x)$
- $x \longleftrightarrow \neg y \equiv (\neg x \vee \neg y) \wedge (y \vee x)$

Démonstration. Tables de vérités passablement ennuyantes ■

- 3SAT est dans NP car on peut prendre les mêmes certificats et la même fonction de vérification que pour SAT
- On prend une formule de SAT, on va la transformer en une entrée de 3SAT
Soit F formules \rightarrow formules X variables. On définit F inductivement par :
 - $F(\varphi_1 \wedge \varphi_2)$:
let $x = \text{newvar}()$
let $(\varphi'_1, y_1) = F(\varphi_1)$ and $(\varphi'_2, y_2) = F(\varphi_2)$ in
 $(x, \varphi'_1 \wedge \varphi'_2 \wedge (x \wedge (\neg x \vee y_1) \wedge (\neg x \vee y_2) \wedge (\neg y_1 \vee \neg y_2 \vee x)))$
 - Pour $F(\varphi_1 \vee \varphi_2)$ on calcule $(\varphi'_1, y_1) = F(\varphi_1)$ and $(\varphi'_2, y_2) = F(\varphi_2)$. On pose $F(\varphi_1 \vee \varphi_2) = \dots$
Désolé $f(x)$ j'ai pas la foi de finir

Définition 4.22. Une formule 3pileSAT est une FNC où chaque clause a exactement 3 littéraux. On note \mathcal{F}_3 l'ensemble de ces formules.

Proposition 4.23. *3PILESAT est NP-complet.*

Démonstration. Méthodiquement :

- 3PILESAT est dans NP d'après le lemme 4.16 appliqué à SAT et \mathcal{F}_3 .
- On se ramène à 3SAT puisque. On considère une formule F de 3SAT. On écrit

$$F = \bigwedge_{i \in I} C_i$$

où, pour tout $i \in I$, la clause C_i s'écrit

$$l_i \quad \text{ou bien} \quad l_{i,1} \vee l_{i,2} \quad \text{ou encore} \quad l_{i,1} \vee l_{i,2} \vee l_{i,3}.$$

Considérons alors pour tout $i \in I$,

$$C'_i := \begin{cases} l_i \vee l_i \vee l_i & \text{si } C_i = l_i \\ l_{i,1} \vee l_{i,2} \wedge l_{i,1} & \text{si } C_i = l_{i,1} \wedge l_{i,2} , \\ C_i & \text{sinon} \end{cases}$$

puis posons

$$F' := \bigwedge_{i \in I} C'_i.$$

Montrons désormais que F est satisfiable si et seulement si F' l'est. Soit ρ une valuation qui satisfait F , alors $\forall i \in I, \rho(C_i) = 1$. Soit $i \in I$, plusieurs cas se distinguent alors :

- $C_i = l_i$, alors $\rho(l_i) = 1$ et par conséquent $\rho(C'_i) = 1$.
- $C_i = l_{i,1} \vee l_{i,2}$, alors $\rho(C'_i) = \max\{\rho(l_{i,1} \vee l_{i,2}), \rho(l_{i,1})\} = 1$.
- $C_i = C'_i$, alors $\rho(C'_i) = \rho(C_i) = 1$.

Dans tous les cas $\rho(C'_i) = 1$ c'est donc que $\rho(F') = 1$. Réciproquement si ρ satisfait F' alors de même par disjonction de cas, on montrerait que $\forall i \in I, \rho(C_i) = 1$ de sorte que $\rho(F) = 1$. On a transformé en temps polynomial une instance de 3SAT en une instance de 3PILESAT telle que

$$\forall F \in \mathcal{F}_3, \quad 3\text{SAT}(F) = \top \iff 3\text{PILESAT}(F') = \top.$$

On vient donc de montrer que $3\text{SAT} \leq_P 3\text{PILESAT}$.

Les conditions du Théorème 4.18 sont remplies, par conséquent 3PILESAT est NP-complet. ■

4.2.2 Exemples dans les graphes (HP)

Définition 4.24. Soit G un graphe non orienté et $k \in \mathbb{N}$, on considère $\text{CLIQUE}(G, k)$ répondant à la question "existe-t-il un sous-graphe de G à k sommets tel que celui-ci soit une clique?".

Proposition 4.25. CLIQUE est **NP-complet**.

Démonstration. Considérons le certificat donné par une liste de k sommets de G formant une clique de taille k . On suppose G donné par un tableau de taille $n := |G|$ (dont le contenu de chaque case est un tableau ou une liste). Si on a un certificat pour une entrée (G, k) qui possède une clique de taille k , alors nécessairement $k \leq n$ donc le certificat est bien polynomial en la taille de G . L'algorithme de vérification est donné par

- Si le nombre de sommets du certificat est plus grand que k ou n , échouer.
- Si les sommets ne sont pas deux à deux distincts, échouer
- Pour chaque $i \in \text{Certificat}$, pour chaque $j \in \text{Certificat} \setminus \{i\}$ tester si $\{i, j\}$ est une arête de G sinon échouer.

dont la complexité s'évalue grossièrement comme

$$O(\min\{n, k\} + k^2 + k^2 n),$$

qui est bien polynomial, donc $\text{CLIQUE} \in \mathbf{NP}$. Montrons à présent qu'il est complet. On se ramène à 3PILESAT . Soit donc

$$F := \bigwedge_{i \in I} C_i$$

(où $\forall i \in I$, $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$) une entrée de 3PILESAT . Construisons un graphe G à partir de F de la manière suivante. pour chaque $i \in I$ on ajoute au graphe trois sommets :

- Un pour chaque littéral $(l_{i,j})_{j \in [1,3]}$.

puis l'on place une arête entre deux sommets si les deux conditions

1. S'ils ne viennent pas de la même clause.
2. qu'ils ne soient pas associés à deux littéraux contraires.

sont satisfaites. Dans la suite on note p le cardinal de I . La taille du graphe est alors $3\#p$ or $\#I \leq |F|$ donc $|G| \leq 9|F|^2$ (polynomial) d'où G est calculé en temps polynomial. On demande de calculer $\text{CLIQUE}(G, p)$. Montrons que

$$\text{CLIQUE}(G, p) = \top \iff 3\text{PILESAT}(F) = \top.$$

S'il existe ρ un témoin de satisfiabilité de F . Pour tout $i \in I$, il existe donc $j \in [1, 3]$ tel que $\rho(l_{i,j}) = 1$. Donc, les $(l_{i,j})_{(i,j) \in I \times [1,3]}$ ne sont pas deux à deux opposés. Les $(l_{i,j})_{(i,j) \in I \times [1,3]}$ sont dans des clauses deux à deux distinctes donc leurs sommets dans G sont reliés et forment donc une clique de taille p . Réciproquement s'il existe une clique de taille p dans G , montrons que F est satisfiable. Les sommets de G sont partitionnés en p indépendants, qui viennent chacun d'une clause. Donc si on a une clique à p sommets, on a pris exactement chaque indépendant associé à une clause. Ces sommets donnent des littéraux l_1, \dots, l_p . On définit ρ par

$$\forall i \in [1, p], \quad \begin{cases} 1 & \text{si } l_i = x_i \\ 0 & \text{sinon} \end{cases}$$

qui est bien définie car si deux littéraux l et l' sont associés à la même variable x alors $l \sim l'$ est une arête du graphe donc $l = l' = x$ ou bien $l = l' = \bar{x}$. Soit $i \in I$, de C_i on tire alors l_i . Par définition de ρ , on a $\forall i \in [1, p]$, $\rho(l_i) = 1$ de sorte que $\forall i \in [1, p]$, $\rho(C_i) = 1$ et par conséquent $\rho(F) = 1$. ■

Voilà une question naissante du public. On fixe $k \in \mathbb{N}$, et pour tout G graphe non orienté on considère le problème de décision $\text{KCLIQUE}(G)$ répondant à la question " G contient une clique de taille k ". KCLIQUE est-il **NP-complet**? On en sait rien, mais on peut démontrer que $\text{KCLIQUE} \in \mathbf{P}$.

Définition 4.26. Soit $G := (V, E)$ un graphe. Une couverture par sommets (*vertex cover* dans la langue de Shakespeare) est une partie V' de V telle que tout arête de E est adjacente à un sommet de V' . Soit alors $k \in \mathbb{N}$, G un graphe non orienté et considérons le problème de décision $\text{VERTEXCOVER}(G, k)$ répondant à la question "existe-t-il une couverture avec k sommets?"

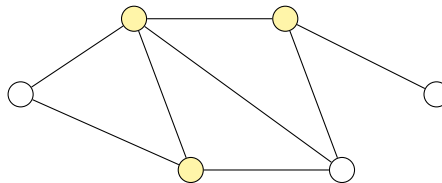


FIGURE 4.1 – Couverture par sommets

Lemme 4.27. Soit $G := (V, E)$. G a une couverture de cardinal k si et seulement si G possède un indépendant de taille $|V| - k$ ce qui équivaut à dire que \overline{G} possède une clique de taille $|V| - k$

Démonstration. Dans les conditions de l'énoncé notons n le cardinal de V .

Si \overline{G} a une clique de cardinal $n - k$. Soit e une arête de G . $e \notin \overline{G}$ donc e n'est pas entre deux sommets de la clique de cardinal $n - k$. Donc e est adjacente à l'un des autres k sommets. On choisit ces k sommets : ils forment bien une couverture.

Réciproquement, si on a une couverture de G à k sommets. On prend les $n - k$ autres sommets. Dans G , ces sommets ne sont pas reliés entre eux. Une arête entre deux de ces sommets ne serait pas couverte. Donc cette arête est dans \overline{G} . C'est vrai pour toute arête entre deux sommets parmi les $n - k$ qui ne sont pas dans la couverture. C'est une clique de taille $n - k$ dans \overline{G} ■

Proposition 4.28. $VERTEXCOVER \in \mathbf{NP}$.

Démonstration. La transformation

$$(G, k) \mapsto (\overline{G}, n - k)$$

est bijective (c'est une involution). Par suite, du lemme précédent on déduit que l'application réciproque est une réduction à SAT, mais cette dernière est clairement polynomiale. D'après le Théorème de réduction, puisque CLIQUE est \mathbf{NP} -complet, on peut alors conclure que VERTEXCOVER est \mathbf{NP} -complet, ce qui achève la preuve. ■

Exemple (Exercice 2 du TD : Circuit Hamiltonien). On prend ce gadget :

On considère une formule de 3PILESAT : $F = \bigwedge_{i=1}^n C_i$ (de taille 3). On note v_1, \dots, v_p les variables de F . Pour chaque variable, on crée un sommet. Pour chaque clause $C_i = l_1 \vee l_2 \vee l_3$, on crée un gadget où e_1 et s_1 sont associés à l_1 , e_2 et s_2 à l_2 et e_3 et s_3 à l_3 .

Pour chaque variable v , on considère les clauses C_{i_1}, \dots, C_{i_q} où $i_1 < \dots < i_q$ où v apparaît positivement. On ajoute des arêtes

on met les deux gadgets ici

On met une arête : la sortie de C_{i_1} associée à v vers l'entrée de C_{i_2} associée à v etc...

Pour C_{i_q} , la sortie associée à v a une arête vers le sommet de la variable suivante. On fait de même avec les clauses où v apparaît sous la forme du littéral négatif \bar{v}

Arête de v vers l'entrée de la première clause. Sortie de v de la première \rightarrow entrée v de la deuxième etc. Sortie de la dernière vers l'entrée de la variable suivante.

Remarque. Chaque sommet associé à une variable a 1 ou 2 arêtes sortantes (une positive, une négative).

Proposition 4.29. F est satisfiable si et seulement si ce graphe a un chemin hamiltonien.

Exemple. Considérons la formule $(a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee c)$, que l'on transforme de sorte à obtenir la figure suivante

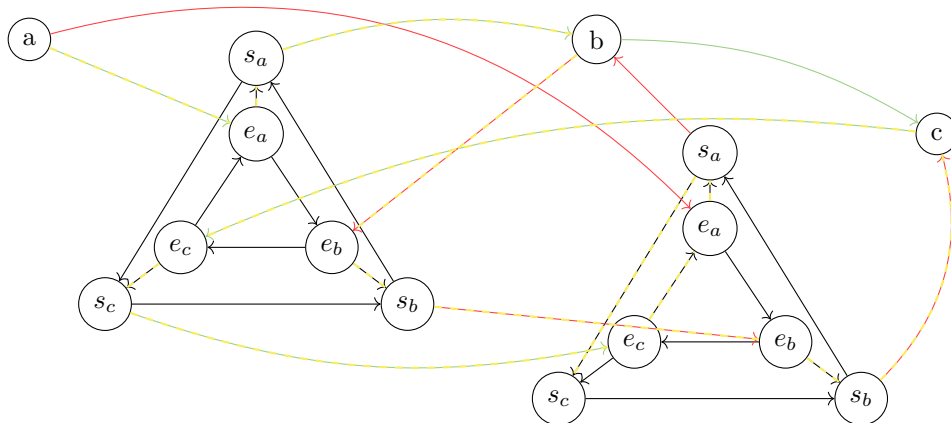


FIGURE 4.2 – La représentation de $(a \vee \bar{b} \vee c) \wedge (\bar{a} \vee \bar{b} \vee c)$

Proposition 4.30. *Chemin Hamiltonien Orienté est \mathbf{NP} -complet*

Proposition 4.31. *Chemin Hamiltonien non orienté est \mathbf{NP} -complet*

Démonstration. <https://a.co/d/aPSFeor>

Soit G un graphe orienté. On lui applique la transformation ci-dessus. Ainsi,
Il existe un cycle hamiltonien dans le nouveau graphe si et seulement ssi G a un cycle hamiltonien. ■

Proposition 4.32. *Cycle hamiltonien est NP-complet*

Algorithme 4.1 Réduction de 3-SAT à Chemin hamiltonien

Pour $(i, j) \in V^2, i \neq j$, **faire** :
 $G_{i,j} = G \cup$ l'arête (i, j)
 $G_{i,j}$ a un cycle hamiltonien ?
si oui renvoyer vrai
Fin pour
renvoyer faux

Démonstration. <https://a.co/d/aPSFeor>

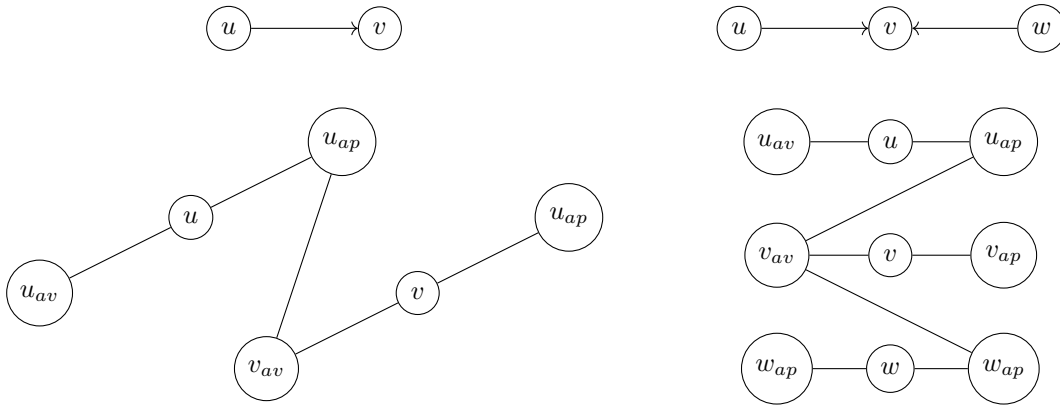


FIGURE 4.3 – Gadget pour transformer un graphe orienté en non orienté

Soit G un graphe orienté. On lui applique la transformation ci-dessus. Ainsi,
Il existe un cycle hamiltonien dans le nouveau graphe si et seulement ssi G a un cycle hamiltonien. ■

4.3 Autres classes de complexité (HP)

Nous avons déjà rencontré deux classes célèbres de complexités que sont **P** et **NP** mais il en existe bien d'autres en fonctions de différents critères que l'on peut demander à la résolution d'un problème de décision. Une bonne partie de la recherche actuelle dans le domaine de la théorie de la complexité est tournée vers l'étude des différentes inclusions entre les différentes classes de complexités (on citera à titre d'exemple le célèbre problème **P = NP**).

Définition 4.33 (PSPACE). On désigne par **PSPACE** l'ensemble des problèmes de décision dont la complexité en mémoire est polynomiale.

Définition 4.34 (EXPTIME). On désigne par **EXPTIME** l'ensemble des problèmes de décision dont la complexité temporelle est exponentielle.

Définition 4.35 (LOGSPACE). On désigne par **LOGSPACE** l'ensemble des problèmes de décision dont la complexité en mémoire est polynomiale en un logarithme de la taille de l'entrée.

Il faut comprendre la définition qui suit dans le contexte de la classe **NP**

Définition 4.36 (CoNP). On désigne par **CoNP** l'ensemble des problèmes de décision dont la vérification des instances négatives est polynomiale.

Proposition 4.37. *On dispose de l'inclusion $P \subseteq CoNP$.*

Démonstration. Se traite de la même manière que $\mathbf{P} \subseteq \mathbf{NP}$.

Bien que ces deux derniers résultats ont déjà été entrevues dans le chapitre, revenons sur une étude des plus élémentaires sur les classes introduites précédemment. Pour cela, commençons par une observation.

Remarque. Un algorithme qui a besoin de k bits de mémoire terminant effectue au plus 2^k étapes de calcul

De cela on déduit deux choses

Proposition 4.38. *On dispose des deux inclusions $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$ ainsi que $\mathbf{LOGSPACE} \subseteq \mathbf{P}$*

Démonstration. c.f remarque ci-dessus.

Il faut additionner au résultat précédent quelques inclusions, celle de la proposition 4.11, celle assez évidente $\mathbf{P} \subseteq \mathbf{EXPTIME}$. On résume assez bien la situation sur un grand diagramme de Venn à l'instar des classiques inclusions $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$, les relations entre les différentes classes de complexité, seulement il y a beaucoup plus de trous. Comme mentionné au début de paragraphe, la recherche dans ce domaine est loin de le comprendre très profondément si bien qu'une carrière brillante en tant que chercheur en informatique fondamentale (que l'on souhaite de bon cœur au lecteur) serait de démontrer des égalités ou des inclusions strictes. Par conséquent, la figure qui suit pourrait très bien être fausse, ou incomplète.

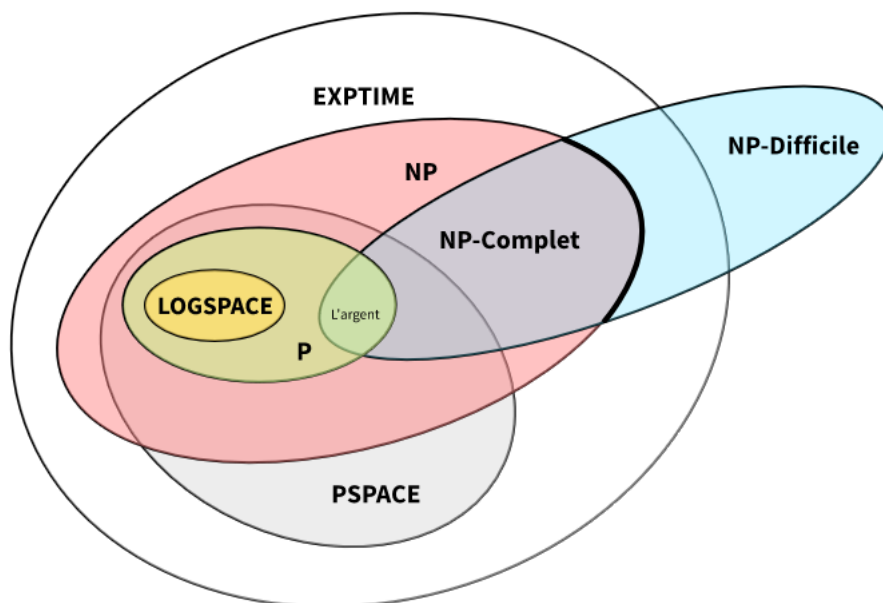


FIGURE 4.4 – Diagramme de Venn des classes de complexité

4.4 Décidabilité

4.4.1 Histoire sympa

Imaginons que l'on se donne deux jeux de dominos. En les combinant (on autorise la réutilisation de dominos déjà utilisés) on crée des mots. Un petit jeu naît de cette observation, peut-on trouver un moyen de former un mot commun à l'aide de ces deux jeux ? Par exemple considérons les deux jeux

- A, AB, BBA
- BAA, AA, BB

Le mot "BBAA" a une correspondance. D'une part,

BBA	A
-----	---

et et d'autre part

BB	AA
----	----

Une manière plus formelle de décrire le problème est de se donner un alphabet Σ ainsi que des mots $u_1, \dots, u_n, v_1, \dots, v_m$ sur celui-ci. La question devient donc : existe-t-il deux suites finies $(i_k)_{1 \leq k \leq p}$ et $(j_l)_{1 \leq l \leq q}$ tel que

$$u_{i_1} u_{i_2} \cdots u_{i_p} = v_{j_1} v_{j_2} \cdots v_{j_q}$$

On note PCP ce problème que l'on appelle correspondance de POST. Faisons une tentative naïve de résolution :

Algorithme 4.2 Algorithme qui recherche une correspondance de POST

Pour chaque couple (u_i, v_j) , **faire** :

Poser $u_i v_j$ à la suite de ce qui a été posé

Si on a une correspondance

renvoyer vrai

Sinon récursivement ajouter d'autres tuiles

Fin pour

On se convainc aisément que quitte à explorer par longueur croissante, l'algorithme renvoie vraie s'il existe une correspondance de POST. Mais s'il n'existe pas de correspondance, il bouclera éternellement. Cet écueil est assez général puisque l'on dispose du théorème suivant (naturellement admis) :

Théorème 4.39. *Quel que soit l'algorithme que vous écrirez, il ne résout pas PCP.*

Le problème PCP est qualifié, en ce sens, de semi-décidable c'est à dire que l'on peut conclure vrai en temps fini, mais pas faux.

4.4.2 Définition

Définition 4.40. Un problème de décision est décidable s'il existe un algorithme (en C, en OCaml, en machine de Turing) qui résout ce problème (peu importe le temps qu'il mettra).

Théorème 4.41 (Admis). *Il existe des problèmes non décidables.*

Exemple. Le problème de l'arrêt. Considérons qu'il existe une fonction OCaml `se_termine : (() -> 'a) -> bool` tel que pour toute fonction `f : () -> 'a`, `se_termine f = true` si et seulement si `ocamlf ()` termine.

```
let rec lol () =
  if se_termine lol
  then lol()
  else 42
```

`lol` se termine si et seulement si elle ne termine pas.

La fonction termine n'existe pas

Proposition 4.42. `se_termine : string -> string -> bool` telle que `se_termine code_f code_entree` décide si `f entree` se termine n'existe pas.

Nous allons décrire plus précisément le phénomène mis en exergue par l'exemple précédent.

Définition 4.43 (Problème de l'arrêt). Le *problème de l'arrêt* et le problème qui répond à la question suivante :

- Entrée : code d'une fonction `f`, code d'une entrée `e` de `f`
- Sortie : est-ce-que le calcul de `f(e)` termine?

Démolissons directement nos maigres espoirs.

Théorème 4.44. *Le problème de l'arrêt est indécidable.*

Esquisse de preuve. Tout repose sur le fait que si `code_f` est le code d'une fonction $f: 'a \rightarrow 'b$. Alors il existe un algorithme `compile : string -> 'a -> 'b` tel que `compile code_f = f`.

S'il existe une fonction

`arrêt : string -> string -> bool`

qui résout le problème de l'arrêt, considérons alors le code de la fonction

```
let diagonale codef =
  let f = compile codef in
  if arret codef codef then f codef
  else 42
```

que l'on place dans une variable `code_diag`. Que vaut alors l'appel `arret code_diag code_diag`?

- Si celui-ci vaut `true`, c'est donc que `diagonale code_diag` s'arrête si bien que lorsque l'on exécute `diagonal code_diag` on entre dans le `then` qui rappelle `diagonal`. Par conséquent `diagonal` ne pourrait s'arrêter, c'est contradictoire.
- Si celui-ci vaut `false` alors c'est la section du `else` qui s'exécute mais alors termine, c'est une nouvelle fois absurde.

Les deux cas examinés mènent à une contradiction, la fonction `arrêt` ne peut donc exister. ■

Remarque. Tout repose sur l'existence d'un compilateur, c'est à dire d'un algorithme qui est capable d'exécuter n'importe quel algorithme.

Exercices

Exercice 4.1 (Somme de sous ensembles). 1. Le problème **2-PARTITION** prend en entrée une liste d'entiers l et décide s'il existe deux sous listes l_1 et l_2 telles que $1 = 11 \oplus 12$ et l_1 et l_2 sont de même somme. Montrer qu'il est **NP-complet**.

2. Le problème **SUBSET-SUM** prend en entrée une liste d'entiers l , un entier N et décide s'il existe une sous liste de l dont la somme vaut N . Montrer qu'il est dans **NP**.

3. On considère une instance de **3SAT** dont les clauses sont C_0, \dots, C_{m-1} et les variables sont x_0, \dots, x_{n-1} . On rappelle en particulier que chaque clause peut être satisfaite par un, deux ou trois littéraux. On définit $b_{ij} \in \{0, 1\}$ qui vaut 1 si et seulement si la variable x_i apparaît dans la clause C_j .

On définit de même des $b'_{i,j}$ pour les littéraux \bar{x}_i . On définit les entiers : $v_i = 10^{m+i} + \sum_{j=0}^{m-1} b_{i,j} 10^j$,

$v'_i = 10^{m+i} + \sum_{j=0}^{m-1} b'_{i,j} 10^j$, $s_j = 10^j$ et $s'_j = 2 \times 10^j$, pour chaque i (variables) et j (clauses).

Déterminer un entier objectif N tel qu l'on vient de définir une instance de **SUBSET-SUM** qui renvoie

4. Montrer que **SUBSET-SUM** est **NP-complet**.

5. Déterminer un algorithme résolvant **SUBSET-SUM** en temps polynomial en fonction de $|l|$ et de N . (On dit que **SUBSET-SUM** est *faiblement NP-complet*.)

6. Le problème **3SUM** prend en entrée une liste de n entiers et décide s'il existe trois entiers de cette liste dont la somme est nulle. Déterminer un algorithme qui résout **3SUM** en temps $O(n^2)$. (Jusqu'en 2014, on pensait que cette complexité était optimale. Ce n'est pas le cas.)

Exercice 4.2 (Problèmes **NP-complets** usuels). Prouver que chacun des problèmes suivants est **NP-complet**. Évaluer la taille de l'entrée de chacun de ces problèmes. Ce sont des exemples de référence qu'il vaut mieux connaître.

Clique Soit G un graphe non orienté et k un entier. Existe-t-il une clique de taille k dans G ?

Indépendant Soit G un graphe non orienté et k un entier. Existe-t-il un indépendant de taille k dans G ?

Couverture par sommets Soit $G = (V, E)$ un graphe non orienté et k un entier naturel. Existe-t-il une partie $V' \subset V$ de cardinal k telle que pour toute arête $e \in E$ il existe un sommet $v \in V'$ tel que $e \in v'$?

Circuit hamiltonien Soit G un graphe non orienté. Un circuit hamiltonien est un cycle élémentaire qui passe par tous les sommets du graphe. Existe-t-il un circuit hamiltonien dans G ? (Plus difficile.)

Coloration Soit G un graphe non orienté et k un entier naturel. Existe-t-il une coloration de G avec au plus k couleurs? (Réduire depuis **3SAT**.)

3-Coloration Soit G un graphe non orienté. Existe-t-il une coloration de G avec au plus 3 couleurs? (Attention, il faut être rigoureux.)

3-Coloration planaire Soit G un graphe non orienté *planaire*. Existe-t-il une coloration de G avec au plus 3 couleurs? (La difficulté est de trouver un gadget planaire. C'est une astuce, difficile sans aide. Une indication est au verso.)

Exercice 4.3 (Pot-pourri de **NP**-complétude). Prouver que chacun des problèmes suivants est **NP**-complet. (Il n'y a aucune chance que nous ne traitions tous ces exemples en classe, cet exercice est donné uniquement pour s'amuser.)

Deux cliques Soit G un graphe non orienté et $K \in \mathbb{N}^*$, existe-t-il deux cliques disjointes de taille K dans G ?

Chevaliers de la table ronde Soient n chevaliers. On connaît un ensemble de paires de chevaliers, féroces ennemis entre eux. Peut-on placer les chevaliers sur une table circulaire de telle sorte qu'aucune paire de féroces ennemis ne soit côte à côte?

Charpentier Soient n baguettes rigides de longueur entières a_1, a_2, \dots, a_n . Les baguettes peuvent être reliées dans cet ordre, bout à bout, par des charnières. Soient k un entier. Peut-on assembler les baguettes de manière qu'en repliant la chaîne obtenue la longueur ne dépasse pas k ?

Clique pour graphe régulier Soient G un graphe dont tous les sommets sont de même degré, et $k \geq 1$ un entier. Existe-t-il une clique de taille supérieure ou égale à k ?

Chemin avec paires interdites Soient $G = (V, E)$ un graphe orienté, deux sommets de ce graphe $s, t \in V$ et une liste $C = \{(a_1, b_1), \dots, (a_n, b_n)\}$ de paires de sommets de G . Existe-t-il un chemin orienté de s vers t dans G qui contient au plus un sommet de chaque paire de la liste C ? (Indication : faire une réduction à partir de **3-SAT**.)

Couverture par sommets avec degré pair Soient G un graphe dont tous les sommets sont de degré pair, et $k \geq 1$ un entier. Existe-t-il un ensemble de sommets de G couvrant toutes les arêtes et de taille inférieure ou égale à k ?

Roue Étant donné un graphe $G = (V, E)$ et un entier $K \geq 3$, déterminer si G contient une roue de taille K , i.e., un ensemble de $K+1$ sommets w, v_1, v_2, \dots, v_K tels que $(v_i, v_{i+1}) \in E$ pour $1 \leq i < K$, $(v_K, v_1) \in E$ et $(v_i, w) \in E$ pour $1 \leq i \leq K$ (w est le centre de la roue).

Dominateur Soit $G = (V, E)$ un graphe non orienté et un entier $K \geq 3$, déterminer si G contient un dominateur de cardinal K , i.e., une partie $D \subset V$ de cardinal K telle que pour tout sommet $u \in V \setminus D$, il existe $u \in D$ avec $(u, v) \in E$.

2-Partition-Approx Soit n entiers a_1, a_2, \dots, a_n , existe-t-il une partie $I \subset \llbracket 1, n \rrbracket$ telle que :

$$\left| \sum_{i \in I} a_i - \sum_{i \notin I} a_i \right| \leq 1.$$

Exercice 4.4 (SAT-n). On appelle **SAT-n** le problème **SAT** restreint aux formules qui n'ont pas plus de n occurrences de la même variable.

1. Montrer que **SAT-3** est au moins aussi dur que **SAT**. En déduire que si $n \geq 3$, **SAT-n** est **NP**-complet.
2. Soit x une variable apparaissant dans une formule F de **SAT-2**. Trouver une formule satisfiable si et seulement si F l'est, dans laquelle x n'apparaît plus.
3. Déterminer la classe de complexité de **SAT-2**.

Exercice 4.5 (Autres classes de complexité).

1. Un problème de décision ϕ est dans la classe **PSPACE** lorsqu'il existe un polynôme P et un algorithme qui calcule, pour toute e , $\phi(e)$ en temps $O(P(|e|))$. Montrer que **NP** \subset **PSPACE**.
2. On dit qu'un problème de décision ϕ est dans la classe **LOGSPACE** lorsqu'il existe un algorithme qui calcule ϕ en réalisant un nombre d'écritures en mémoire borné par $O(\log(|e|))$ pour chaque entrée e . On n'impose aucune borne sur le nombre de lectures. Montrer l'inclusion **LOGSPACE** \subset **P**.

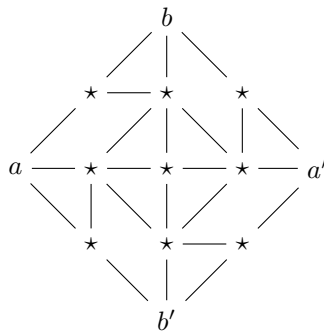


FIGURE 4.5 – Gadget pour la 3 coloration planaire

Chapitre 5

Automates

Sommaire

5.0	Introduction	75
5.1	Automates finis déterministes	76
5.1.1	Définitions	76
5.1.2	Langage d'automate	77
5.1.3	Lemme de l'étoile	79
5.2	Autour des automates déterministes et études linguistiques	80
5.3	Automates non déterministes	80
5.3.1	Définition	80
5.3.2	Automates à transition spontanée	82
5.4	Opérations sur les langages reconnaissables	83
5.5	Théorème de KLEENE	85
5.5.1	Le théorème	85
5.5.2	Algorithme de BERRY-SETHI	85
	Exercices	85

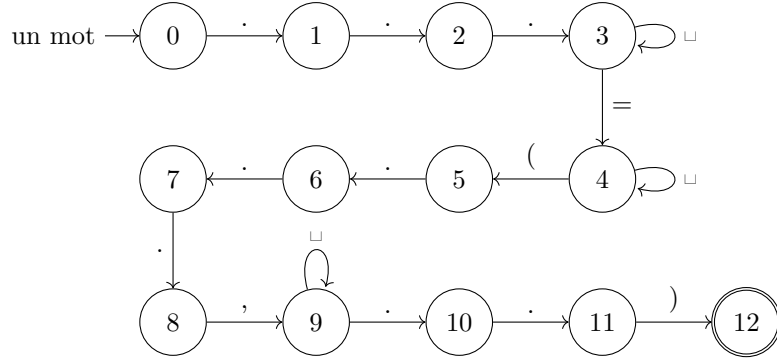
Tout est automate... presque tout

F.Hatat

L'annexe hors programme sur les machines de Turing est une parfaite introduction aux automates qui dans l'idée fonctionnent de manière comparable, mais qui sont néanmoins plus faibles.

5.0 Introduction

Faisons un léger retour sur les expressions régulières. Nous souhaitons identifier des chaînes de caractères se présentant de la forme $[MOT\ 1] = ([MOT\ 2], [MOT\ 3])$ où $[MOT\ 1]$, $[MOT\ 2]$ et $[MOT\ 3]$ correspondent à une suite quelconque de trois caractères (e.g. : $AAA = (ABC, ABZ)$). Pour ce faire, on peut utiliser l'expression régulière $\dots \sqcup^* = \sqcup^* (\dots, \sqcup^* \dots)$, dont on peut représenter l'action par le graphe représenté ci-dessous.



où l'on passe d'un état à un autre si le caractère que l'on lit remplit la condition symbolisée au dessus de la flèche. Ainsi, un mot est reconnu par l'expression régulière s'il on atteint le noeud numéro 12. C'est l'idée sous-jacente à un automate que l'on développe dans la prochaine partie.

5.1 Automates finis déterministes

5.1.1 Définitions

Définition 5.1 (Automate déterministe). Un *automate fini déterministe* est un quintuplet $(\Sigma, Q, q_I, F, \delta)$ où

- Σ est un alphabet,
- Q est un ensemble fini (ensemble des états),
- $q_I \in Q$ est nommé état initial,
- $F \subset Q$ c'est l'ensemble d'états finaux (aussi dit acceptants),
- $\delta : \mathcal{D} \rightarrow Q$, avec $\mathcal{D} \subseteq Q \times \Sigma$ est appelé fonction de transition.

Définition 5.2 (Automate complet). dans la définition précédente, si l'ensemble de définition \mathcal{D} de la fonction de transition δ est $Q \times \Sigma$ on qualifiera l'automate de *complet*.

Pour le reste de ce chapitre, on se donne \mathcal{A} un automate déterministe $(\Sigma, Q, q_i, F, \delta)$ que l'on suppose dépourvue de quelconque propriété.

Définition 5.3. On appelle graphe d'un automate \mathcal{A} le graphe orienté étiqueté dont l'ensemble des sommets est Q . On décrète ensuite que deux états sont reliés si et seulement s'il existe une transition entre ces derniers, la lettre qui a donné naissance à cette transition figurera comme étiquette de l'arête.

Remarque. Ce graphe décrit donc l'action de l'automate et par conséquent peut tenir de définition pour la fonction de transition, des états finaux et de l'état initial. Dans les futurs graphes, un sommet coloré de

- vert muni d'une flèche \rightarrow représentera un sommet initial,
- bleu représentera un état quelconque,
- rouge et de contour doublé représentera un sommet d'état final,
- jaune muni d'une flèche \rightarrow et d'un contour doublé, représentera un sommet à la fois final et initial.

Définition 5.4 (δ^*). On définit $\delta^* : Q \times \Sigma^* \rightarrow Q$ l'extension aux mots de δ par récurrence sur la longueur de ceux-ci. Pour tout $(q, u) \in Q \times \Sigma^*$,

$$\delta^*(q, u) := \begin{cases} q & \text{si } u = \varepsilon \\ \delta^*(\delta(q, a), w) & \text{s'il existe } (a, w) \in \Sigma \times \Sigma^* \text{ tel que } u = a \cdot w \end{cases}$$

Remarque. la fonction δ^* effectue intuitivement un parcours dans \mathcal{A} vers un certain état.

5.1.2 Langage d'automate

Définition 5.5 (Mot accepté). Soit $u \in \Sigma^*$. On dit que u est accepté par \mathcal{A} si $\delta^*(q_I, u)$ existe avec $\delta^*(q_I, u) \in F$.

Proposition 5.6. Un mot $u \in \Sigma^*$ est accepté si et seulement si il existe un chemin dans \mathcal{A} étiqueté par u dont le premier sommet est q_I et le dernier est dans F (que l'on nommera dans la suite chemin acceptant).

Démonstration. On se donne $u \in \Sigma^*$ et l'on raisonne par double implication.

\Rightarrow Si u est accepté, notons u_1, \dots, u_n ses lettres, alors $\delta^*(q_I, u)$ termine et est à valeur dans F . Par suite, définissons par récurrence pour tout $k \in \llbracket 0, n-1 \rrbracket$

$$q_k := \begin{cases} q_I & \text{si } k = 0 \\ \delta^*(q_I, u_1 \cdots u_k) & \text{sinon} \end{cases}$$

et observons alors que par définition de δ^*

$$\forall k \in \llbracket 0, n-1 \rrbracket, \quad q_{k+1} = \delta(q_k, u_{k+1}).$$

Par conséquent dans \mathcal{A} l'arête $q_k \xrightarrow{u_{k+1}} q_{k+1}$ existe. Mais alors puisque u est accepté, $q_n \in F$ si bien que l'on a le chemin suivant dans \mathcal{A} :

$$q_I = q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} \cdots \xrightarrow{u_n} q_n \in F,$$

qui satisfait aux conditions d'un chemin que l'on cherchait à construire.

\Leftarrow s'il existe un chemin $\gamma : \llbracket 0, n \rrbracket \rightarrow Q$ dans \mathcal{A} étiqueté par u tel que $\gamma_0 = q_I$ et $\gamma_n \in F$. Ainsi, quelque soit $k \in \llbracket 0, n-1 \rrbracket$, $\delta(\gamma_k, u_k) = \gamma_{k+1}$ de sorte qu'une récurrence immédiate fournit alors que $\forall k \in \llbracket 0, n \rrbracket$, $\delta^*(\gamma_0, u_1 \cdots u_k) = \gamma_k$ donc en particulier pour $\delta^*(q_I, u) = \delta^*(\gamma_0, u_1 \cdots u_n) = \gamma_n \in F$ (par hypothèse) si bien que u est accepté par \mathcal{A} .

D'où le résultat. \blacksquare

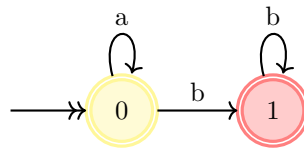
Définition 5.7 (Langage de \mathcal{A}). On appelle langage de \mathcal{A} l'ensemble des mots accepté par \mathcal{A} noté $L(\mathcal{A})$, formellement

$$L(\mathcal{A}) := \{u \in \Sigma^* : u \text{ est accepté par } \mathcal{A}\}.$$

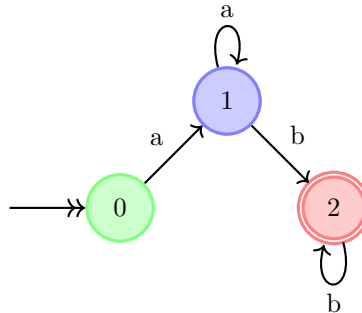
Exemple. Considérons l'automate défini par $\Sigma := \{a, b\}$ puis $Q := \{0, 1\}$ avec $q_i := 0$, $F := \{0, 1\}$ et la fonction de transition

$$\delta : \begin{cases} (0, a) \mapsto 0 \\ (0, b) \mapsto 1 \\ (1, b) \mapsto 1 \\ (1, a) \mapsto 0 \end{cases}$$

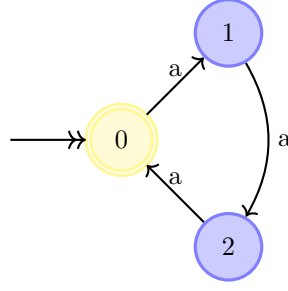
Celui-ci a pour langage $\{a^n b^m\}_{(n,m) \in \mathbb{N}^2}$ et se représente comme suit :



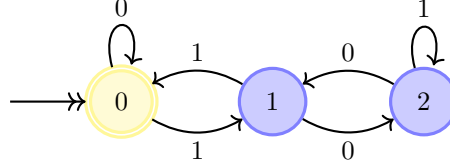
Exemple. le langage $\{a^n b^m\}_{(n,m) \in (\mathbb{N}^*)^2}$ est celui de l'automate



Exemple. le langage $\{a^n\}_{n \in 3\mathbb{N}}$ est celui de l'automate



Exemple. Le langage $\{u \in \{0,1\}^* : \overline{u}^{(2)} \in 3\mathbb{N}\}$ est celui de l'automate :



Définition 5.8. On dit qu'un langage L sur Σ est reconnaissable par un automate fini déterministe lorsqu'il existe un automate fini déterministe \mathcal{A} tel que $L = L(\mathcal{A})$.

Remarque. Cette appellation est assez longue n'est-ce pas ? Lorsque le contexte est suffisamment clair (car parfois nous travaillerons avec des automates de nature différente) nous dirons plus simplement que L est reconnaissable. On notera dans la suite $\text{Rec}(\Sigma)$ (respectivement $\text{Rec}_{\text{complet}}(\Sigma)$) l'ensemble des langages sur Σ reconnaissables par un automate fini déterministe (respectivement complet).

Proposition 5.9. On a l'égalité $\text{Rec}(\Sigma) = \text{Rec}_{\text{complet}}(\Sigma)$.

Démonstration. Le sens $\text{Rec}_{\text{complet}}(\Sigma) \subset \text{Rec}(\Sigma)$ est évident (tout automate complet est en particulier un automate). Pour le sens réciproque, on considère un langage L reconnu par un automate fini déterministe $\mathcal{A} = (\Sigma, Q, q_I, F, \delta)$. On pose $Q_c := Q \cup \{p\}$ où p est un nouvel état. On définit également pour tout $(q, a) \in Q_c \times \Sigma$

$$\delta_c(q, a) := \begin{cases} \delta(q, a) & \text{si } (q, a) \in \mathcal{D} \\ p & \text{sinon} \end{cases}.$$

L'automate $\mathcal{A}_c = (\Sigma, Q_c, q_I, F, \delta_c)$ est complet. Montrons que $L(\mathcal{A}) = L(\mathcal{A}_c)$.

\subseteq Soit $u \in L(\mathcal{A})$. Notons u_1, \dots, u_n ses lettres. Il existe des états q_0, \dots, q_n tels que $q_0 = q_I$, $q_n \in F$ et $\forall i \in \llbracket 0, n-1 \rrbracket, \delta(q_i, u_{i+1}) = q_{i+1}$. Par définition,

$$\forall i \in \llbracket 0, n-1 \rrbracket, \delta_c(q_i, u_{i+1}) = q_{i+1}$$

ce qui montre qu'il existe un chemin acceptant dans \mathcal{A}_c et donc finalement $u \in L(\mathcal{A}_c)$.

\supseteq Soit $u \in L(\mathcal{A}_c)$. On note u_1, \dots, u_n ses lettres. Soit q_0, \dots, q_n un chemin acceptant dans \mathcal{A}_c . Par récurrence descendante, on montre que $\forall k \in \llbracket 0, n \rrbracket, q_k \neq p$.

- Initialisation : q_n est dans F qui est inclus dans Q qui ne contient pas p de sorte que $q_n \neq p$.
- Hérité : soit $k \in \llbracket 0, n-1 \rrbracket$ tel que $q_k \neq p$. Dès lors, si $q_{k-1} = p$ alors $\forall a \in \Sigma, \delta_c(q_{k-1}, a) = p \neq q_k$ ce qui n'est pas compatible avec l'hypothèse préalable donc $q_{k-1} \neq p$. Par conséquent $q_k \xrightarrow{u_{k+1}} q_{k+1}$ est une transition de \mathcal{A} .

En conclusion, on a un chemin acceptant dans \mathcal{A} d'où $u \in L(\mathcal{A})$. ■

Remarque. Cela traduit l'idée que l'ensemble de définition n'est pas une mesure pertinente pour la faculté d'un automate à reconnaître des langages.

Une question naturelle apparaît, tous les langages sont-ils reconnaissables ? Pour répondre à cette question, nous développons un outil tout particulier.

5.1.3 Lemme de l'étoile

Dans cette sous-section, on explore l'un des résultats fondamentaux de la théorie des automates, le lemme de l'étoile aussi dit lemme de pompage.

Lemme 5.10 (Lemme de l'étoile). *Soit L un langage reconnaissable. Il existe un entier N tel que pour tout $u \in L$ tel que $|u| \geq N$, alors il existe x, y, z trois mots tels que*

$$u = xyz, \quad y \neq \varepsilon, \quad |xy| \leq N, \quad \forall k \in \mathbb{N}, \quad xy^k z \in L.$$

Démonstration. Soit $L \in \text{Rec}(\Sigma)$, il existe alors un automate \mathcal{A} tel que $L = L(\mathcal{A})$. On note N le nombre d'états que comporte l'automate \mathcal{A} et (q_0, \dots, q_n) un chemin acceptant pour $u = u_1 \dots u_n$ où $n = |u| \geq N$. Par principe des tiroirs, (q_0, \dots, q_N) contenant $N + 1$ éléments pour N états, il existe $i < j \leq N$ tels que $q_i = q_j$. On pose $x := u_1 \dots u_i$, $y := u_{i+1} \dots u_j$ et $z := u_{j+1} \dots u_n$. On a bien $u = xyz$, $y \neq \varepsilon$ puisque $i + 1 \leq j$ et $|xy| = j \leq N$. De plus, $\delta^*(q_i, y) = q_j = q_i$ donc par récurrence, on démontre que

$$\forall k \in \mathbb{N}, \quad \delta^*(q_i, y^k) = q_i,$$

et ainsi

$$\forall k \in \mathbb{N}, \quad \delta^*(q_i, xy^k z) = q_n.$$

Or q_n est un état acceptant donc $\forall k \in \mathbb{N}, xy^k z \in L$. ■

Remarque. La troisième condition indique que parmi toutes les boucles possibles, le lemme donne la première existante.

Voici quelques applications du lemme de l'étoile

Exemple. Considérons $L := \{u \in \Sigma^* : \exists (n, m) \in \mathbb{N}^2, n \leq m, u = a^n b^m\}$ sur $\Sigma := \{a, b\}$. L n'est pas reconnaissable car en effet, s'il l'était, le lemme de l'étoile donne l'existence de $N \in \mathbb{N}$ tel que pour tout $u \in L$ tel que $|u| \geq N$, alors il existe x, y, z trois mots tels que

$$u = xyz, \quad y \neq \varepsilon, \quad |xy| \leq N, \quad \forall k \in \mathbb{N}, \quad xy^k z \in L.$$

Mais, par définition, le mot $a^N b^N$ est dans L , par conséquent il existe x, y, z trois mots vérifiant la condition précédente. Observons alors que la condition $|xy| \leq N$ entraîne xy préfixe de a^N . Il existe donc $k \in \llbracket 1, N - 1 \rrbracket$ tel que $x = a^k$ et $y = a^{N-k}$ (car $y \neq \varepsilon$). Par conséquent,

$$\forall \ell \in \mathbb{N}, \quad xy^\ell z \in L,$$

en particulier pour $\ell = 2$, on a $u' := a^k a^{2(N-k)} b^N \in L$ ce qui est absurde puisque $|u'|_a = 2N - k > N = |u'|_b$.

En réalité, on peut généraliser les idées développées ci-dessus et obtenir :

Lemme 5.11 (Lemme de l'étoile généralisé (HP)). *Soit L un langage reconnaissable par un automate \mathcal{A} . Il existe un entier N tel que pour tout $u \in L$ qui s'écrit de la forme $u = rst$ où $|s| \geq N$, il existe trois mots x, y, z tel que*

$$s = xyz, \quad |xy| \leq N, \quad y \neq \varepsilon, \quad \forall k \in \mathbb{N}, rxy^k zt \in L.$$

Démonstration. On s'appuie sur le lemme de l'étoile initial. On note $r = r_1 \dots r_n$, $s = s_1 \dots s_m$ et $t = t_1 \dots t_p$ puis q_0, \dots, q_{n+m+p} un chemin acceptant de u . On considère l'automate \mathcal{A}' restreint aux états q_n, \dots, q_{n+m} à partir de \mathcal{A} . On note $L' = L(\mathcal{A}')$. \mathcal{A}' possède $m + 1$ états... ■

Remarque. Cette version permet de "pomper" dans n'importe quel facteur.

Voici quelques applications du lemme de l'étoile généralisé

Exemple. Considérons $L := \{u \in \Sigma^* : \exists (n, m) \in \mathbb{N}^2, n \leq m, u = ca^n b^m\}$ sur $\Sigma := \{a, b, c\}$. L'exemple est très proche d'un autre où l'on avait utilisé le lemme de pompage dans sa version au programme. Ici, s'il on appliquait le susmentionné lemme, il faudrait procéder à un certain nombre de disjonctions de cas. Or avec la version généralisée du lemme de l'étoile, il suffit de pomper dans les facteurs $a^n b^m$ et l'on recourt alors à la même démonstration, pratique non ?

Exemple. Considérons le langage $L := (a^p)_{p \in \mathbb{P}}$. C'est alors une simple question d'arithmétique couplée au lemme de l'étoile généralisé !

On en déduit le théorème suivant par les exemples qui précèdent

Théorème 5.12. *Pour tout langage Σ , l'inclusion $\text{Rec}(\Sigma) \subset \mathcal{P}(\Sigma^*)$ est stricte. Autrement dit, il existe des langages qui ne sont reconnus par aucun automate.*

Poincaré dirait que ce ne sont guère d'honnêtes mathématiciens !

5.2 Autour des automates déterministes et études linguistiques

Définition 5.13. On dit d'un état de \mathcal{A} qu'il est

- accessible s'il existe un chemin de q_i vers celui-ci,
- co-accessible s'il existe un chemin de cet état vers un état final.


Définition 5.14. On dit que \mathcal{A} est

- accessible si tout ses états sont accessibles,
- co-accessible si tout ses états sont co-accessibles,
- émondé s'il est à la fois accessible et co-accessible.

Théorème 5.15. *Un langage L sur Σ est reconnaissable si et seulement s'il est reconnaissable pour un automate accessible, co-accessible ou émondé.*

Démonstration. ■

Remarque. Il découle du théorème précédent que lorsque l'on souhaite démontrer un résultat sur les langages d'automate, alors toutes les propriétés précédentes peuvent être faites sur l'automate sur lequel on travaille, et cela, sans nuire à la généralité du raisonnement. Ceci tient également pour la proposition 5.9.

 Attention, certaines hypothèses sont contradictoires, on ne peut pas supposer à la fois qu'un automate est complet, et en même temps émondé, bien que l'on ai vu que la supposition de l'une de ces hypothèses ne nuit d'aucune manière à la généralité d'un raisonnement fait pour démontrer forces résultats sur les langages d'automate, en revanche le mélange fait mauvais ménage.

5.3 Automates non déterministes

5.3.1 Définition

Définition 5.16. Un *automate fini non-déterministe* est un quintuplet $(\Sigma, Q, I, F, \delta)$ où

- Σ est un alphabet,
- Q est un ensemble fini (ensemble des états),
- $I \subseteq Q$ est nommé ensemble des états initiaux,
- $F \subseteq Q$ c'est l'ensemble d'états finaux (aussi dit acceptants),
- $\delta \subseteq Q \times \Sigma \times Q$ en abusant de la terminologie, que l'on appelle relation de transition.

On définit de manière analogue la fonction δ^* , le graphe d'automate se transforme en le graphe de la relation δ . Ce point de vue en terme de graphe devient alors impératif pour parler de mots acceptés ce qui motive la définition suivante :

Définition 5.17. Un mot $u \in \Sigma^*$ est accepté par un automate fini non déterministe lorsqu'il existe un chemin $\gamma : \llbracket 0, n \rrbracket \rightarrow \delta$ tel que $q_0 \in I$, $q_n \in F$ et

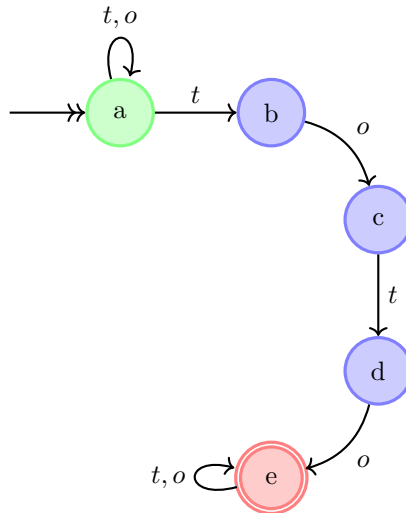
$$\forall i \in \llbracket 0, n-1 \rrbracket, (\gamma_i, u_i, \gamma_{i+1}) \in \delta.$$

(c'est à dire étiqueté par u).

On définit alors de même que pour un automate déterministe la notion de langage d'automate que l'on notera de la même manière. On désignera également $\text{NRec}(\Sigma)$ l'ensemble des langages sur Σ reconnus par un automate fini non déterministe.

La notion d'automate déterministe est un cas particulier d'automate non-déterministe. En effet, on peut voir toute fonction comme son graphe fonctionnel ce qui permet de définir une relation binaire. Par conséquent, la notion d'automate non déterministe prolonge donc celle d'automate déterministe, ce qui justifie par ailleurs l'usage des mêmes notations pour les objets gravitant autour des automates. On pourrait par ailleurs penser que le pouvoir expressif des automates non-déterministe est plus puissant que celui des automates déterministe. Nous allons explorer cette question dans la suite de cette sous-partie.

Exemple (Tautomate). Considérons l'automate



qui reconnaît le langage $\Sigma^* \cdot toto \cdot \Sigma^*$ sur un alphabet Σ contenant les lettres "t" et "o" (d'où son petit nom).

Observons la simplicité de l'automate de l'exemple précédent pour décrire son langage qui pourtant semble si complexe.

Définition 5.18 (Automate des parties). Soit $\mathcal{A} := (\Sigma, Q, I, F, \delta)$ un automate fini non déterministe. On appelle *automate des parties de \mathcal{A}* l'automate $\mathcal{A}_p := (\Sigma, Q_p, q_I, F_p, \delta_p)$ définis par

- $Q_p := \mathcal{P}(Q)$,
- $q_I := I \in \mathcal{P}(Q) = Q_p$,
- $F_p := \{X \in Q_p : X \cap F \neq \emptyset\}$,
- pour tout $(X, a) \in Q_p \times \Sigma$, $\delta_p(X, a) := \{y \in Q : \exists x \in X, (x, a, y) \in \delta\}$.

Qui est un automate fini déterministe et aussi complet.

Remarque. Ce qui motive intuitivement la définition précédente, c'est l'idée que l'on puisse parcourir simultanément toutes les arêtes partant d'un état dans δ .

Cette transformation est particulièrement coûteuse, car l'ensemble des états de l'automate des parties est en taille exponentielle en la taille de l'ensemble des états de l'automate de départ. Dans la pratique, on peut se dispenser d'une bonne partie des éléments de Q_p car l'on préfère dériver l'idée pour construire un automate fini déterministe émondé.

Théorème 5.19. On a l'égalité $\text{Rec}(\Sigma) = \text{NRec}(\Sigma)$.

Démonstration. Par double inclusion :

- \subseteq La première inclusion découle du fait que les automates non déterministes prolongent les automates déterministes (même s'il est vrai que nous n'avons pas démontré que la transformation d'un automate déterministe en automate non déterministe laisse invariant le langage reconnu, on considère ce point évident).
- \supseteq Soit \mathcal{A} un automate non déterministe. Il s'agit de montrer, pour obtenir l'inclusion réciproque, que $L(\mathcal{A}) = L(\mathcal{A}_p)$.
- \subseteq Si $u \in L(\mathcal{A})$, alors il existe un chemin acceptant de u dans \mathcal{A} :

$$I \ni q_0 \xrightarrow{u_0} q_1 \xrightarrow{u_1} \dots \xrightarrow{u_{n-1}} q_n \in F.$$

Définissons récursivement, pour tout $i \in \llbracket 0, n-1 \rrbracket$:

$$X_i := \begin{cases} I & \text{si } i = 0 \\ \delta_p(X_{i-1}, u_{i-1}) & \text{sinon} \end{cases},$$

et montrons que $\forall i \in \llbracket 0, n-1 \rrbracket$, $q_i \in X_i$ par récurrence finie.

- Initialisation : par construction $q_0 \in I = X_0$.

- **Hérédité** : soit $i \in \llbracket 0, n-2 \rrbracket$ tel que $q_i \in X_i$. Or $(q_i, u_i, q_{i+1}) \in \delta$ par hypothèse, si bien que par définition $q_{i+1} \in X_{i+1}$.

Finalement, $q_n \in \delta_p^*(q_I, u)$. Mais $q_n \in F$ donc $X_n \cap F \neq \emptyset$ ce qui montre que $X_n \in F_p$. On vient ainsi d'exhiber un chemin acceptant pour u dans \mathcal{A}_p c'est donc que $u \in L(\mathcal{A}_p)$.

\square Si $u \in L(\mathcal{A}_p)$, il existe un chemin acceptant de u dans \mathcal{A}_p :

$$I = X_0 \xrightarrow{u_0} X_1 \xrightarrow{u_1} \dots \xrightarrow{u_{n-1}} X_n \in F_p.$$

Par récurrence descendante on construit $(q_i)_{0 \leq i \leq n}$ tel que

$$\forall i \in \llbracket 1, n \rrbracket, \begin{cases} q_i \in X_i \\ \delta_p(X_{i-1}, u_{i-1}) = X_i \end{cases}$$

- **Initialisation** : par définition, on peut donc trouver $q_n \in X_n \cap F \neq \emptyset$.
- **Hérédité** : soit $i \in \llbracket 1, n \rrbracket$ tel que les q_i, \dots, q_n soient construit tel que ci-dessus. Par définition de δ_p , il existe $q_{i-1} \in X_{i-1}$ tel que $(q_{i-1}, u_{i-1}, q_i) \in \delta$.

On dispose ainsi d'un chemin acceptant

$$I \ni q_0 \xrightarrow{u_0} q_1 \xrightarrow{u_1} \dots \xrightarrow{u_{n-1}} q_n \in F,$$

pour u dans \mathcal{A} (puisque par construction, les flèches sont des transitions de \mathcal{A}) ce qui montre que $u \in L(\mathcal{A})$. ■

On peut donc répondre à notre interrogation de départ, non les automates non déterministe ne sont pas plus riches linguistiquement que les automates déterministe. Nonobstant, encore une fois ils permettent une plus large souplesse lorsque l'on manipule des questions de langage d'automates car l'on peut désormais faire l'hypothèse, sans perte de généralité aucune, qu'un langage est reconnu par un automate non déterministe.

5.3.2 Automates à transition spontanée

Définition 5.20 (Transitions spontanées). Une transition spontanée notée aussi ε -transition permet à un automate de changer d'état sans consommer un symbole d'entrée.

Définition 5.21 (Automates avec epsilon transitions). On ajoute des ε -transitions, alors $\delta \subset (Q \times \Sigma \times Q) \cup (Q \times Q)$. Un chemin de l'automate est étiqueté par un mot u si ce chemin est de la forme :

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_{n+1}$$

où $a_0 a_1 \dots a_n = u$, en comptant les ε comme des mots vides dans la concaténation.

Définition 5.22 (Mot accepté). Un mot u est accepté par un automate avec epsilon transitions s'il existe un chemin étiqueté par u d'un état initial vers un état final.

Définition 5.23. On appelle ε -chemin d'un automate à transition spontanée tout chemin de la forme

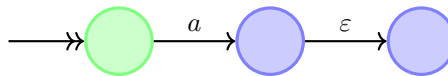
$$q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q_n,$$

que l'on notera dans la suite

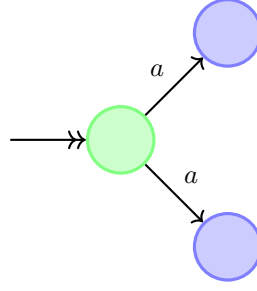
$$q_0 \rightarrow^+ q_n.$$

Définition 5.24. On note $\text{NRec}_\varepsilon(\Sigma)$ les langages acceptés par automate à transitions spontanées.

Exemple. Transfert d'un automate avec ε -transitions vers un automate sans ε -transitions :



en :



Proposition 5.25. $\text{NRec}_\varepsilon(\Sigma) = \text{NRec}(\Sigma)$.

Démonstration. On raisonne encore par double inclusion :

- \subseteq De même que pour la preuve du théorème 5.9 on construit un nouvel automate. Soit $L \in \text{NRec}_\varepsilon(\Sigma)$ reconnu par $\mathcal{A} := (\Sigma, Q, I, F, \delta)$. Définissons :
- $I^+ := I \cup \{q \in Q : \exists q_I \in I, q_I \rightarrow^+ q\}$.
 - δ^+ à partir de δ en remplaçant tout chemin de \mathcal{A} de la forme $q \xrightarrow{a} q' \rightarrow^+ q''$ par une arête $q \rightarrow^+ q''$.
- et considérons alors l'automate $\mathcal{A}^+ := (\Sigma, Q, I^+, F, \delta^+)$ qui est non déterministe. Montrons alors que $L(\mathcal{A}) = L(\mathcal{A}^+)$:
- Soit $u \in L(\mathcal{A})$ que l'on écrit $u_0 \cdots u_{n-1}$, il existe donc un chemin acceptant

$$q_0 \xrightarrow{u_0} q_1 \rightarrow \cdots \xrightarrow{u_{n-1}} q_n$$

dans \mathcal{A} qui est une alternance de ε -chemin si bien qu'il existe $0 \leq j_1, \dots, j_k \leq n$ des entiers tels que $\forall \ell \in [1, k], u_{j_\ell} \neq \varepsilon$ et $u = u_{j_1} \cdots u_{j_k}$. Il existe alors $0 \leq i_1 \leq j_1 < i_2 < j_2 < \cdots < i_k < j_k \leq n$ tel que

$$I \ni q_0 \xrightarrow{\varepsilon^+} q_{i_1} \xrightarrow{u_{j_1}} q_{j_1} \xrightarrow{\varepsilon^+} q_{i_2} \xrightarrow{u_{j_2}} q_{j_2} \xrightarrow{\varepsilon^+} \cdots \xrightarrow{\varepsilon^+} q_{i_k} \xrightarrow{u_{j_k}} q_{j_k} \in F.$$

Par construction de δ^+ et de I^+ on a donc dans \mathcal{A}^+ le chemin

$$I^+ \ni q_0 \xrightarrow{u_0} q_{j_1} \xrightarrow{u_1} \cdots \xrightarrow{u_{j_k}} q_{j_k} \in F$$

qui est donc acceptant pour u dans \mathcal{A}^+ donc $u \in L(\mathcal{A}^+)$.

- La réciproque se traite d'une manière analogue il faut simplement faire attention au fait qu'une arête entre deux états peut passer par un ε -chemin dans \mathcal{A} . d'où le résultat et *a fortiori* $L \in \text{NRec}(\Sigma)$.

- \supseteq L'inclusion réciproque tient du fait que tout automate non déterministe est en particulier un automate non déterministe avec ε -transition.

d'où le résultat. ■

Remarque. Dans la preuve précédente, la construction \mathcal{A}^+ fait un "bond en avant" dans l'automate \mathcal{A} . On aurait également pu considérer un automate de "bond en arrière" \mathcal{A}^- .

Corollaire 5.26. $\text{NRec}_\varepsilon(\Sigma) = \text{Rec}(\Sigma)$.

5.4 Opérations sur les langages reconnaissables

On pourrait aussi qualifier cette section de chirurgie d'automate en ce sens que beaucoup des manipulations que nous ferons pour démontrer les résultats qui suivront seront basés sur la construction (ou modification) d'un automate de départ afin de répondre au problème posé.

Proposition 5.27. Soit $(L_1, L_2) \in \text{Rec}(\Sigma)^2$ alors $L_1 \cap L_2 \in \text{Rec}(\Sigma)$.

Démonstration. Soit $\mathcal{A}_1 := (\Sigma, Q_1, q_1, F_1, \delta_1)$ (respectivement $\mathcal{A}_2 := (\Sigma, Q_2, q_2, F_2, \delta_2)$) reconnaissant L_1 (respectivement L_2) que l'on peut supposer sans perte de généralité aucune complet. La construction proposée ici se nomme "automate produit". L'intuition est qu'il faille parcourir en même temps les mots reconnus respectivement par les deux automates. Posons

- $Q := Q_1 \times Q_2$,
- $q_I := (q_1, q_2)$,
- pour tout $(q_1, q_2) \in Q$ et $a \in \Sigma$,

$$\delta((q_1, q_2), a) := (\delta_1(q_1, a), \delta_2(q_2, a))$$

- $F := F_1 \times F_2$.

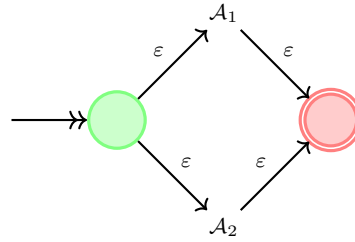
L'automate $\mathcal{A} := (\Sigma, Q, q_I, F, \delta)$ reconnaît $L_1 \cap L_2$. En effet ■

Remarque. Il peut être vraiment utile de retenir la construction de l'automate produit qui peut servir dans bien des situations différentes.

Proposition 5.28. Soit $(L_1, L_2) \in \text{Rec}(\Sigma)^2$ alors $L_1 \cup L_2$ et $L_1 \cdot L_2 \in \text{Rec}(\Sigma)$ sont reconnaissables.

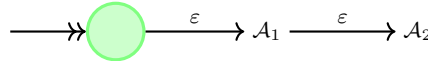
Démonstration. On fera ici l'usage d'automates à ε -transition. Soit \mathcal{A}_1 (respectivement \mathcal{A}_2) reconnaissant L_1 (respectivement L_2).

— L'idée est simplement de formaliser la construction intuitive :



que l'on appelle *construction de Thompson*.

— L'idée est simplement de formaliser la construction intuitive :



■

Proposition 5.29. Pour toute famille finie $(L_i)_{i \in I}$ de langages reconnaissables, les langages

$$\bigcup_{i \in I} L_i, \quad \bigcap_{i \in I} L_i, \quad \prod_{i \in I} L_i$$

sont reconnaissables.

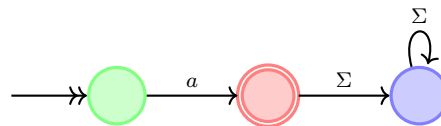
Démonstration. Récurrences immédiates depuis les deux propositions précédentes. ■

Proposition 5.30. On dispose des assertions suivantes

- $\forall a \in \Sigma, \{a\} \in \text{Rec}(\Sigma)$.
- $\{\varepsilon\} \in \text{Rec}(\Sigma)$.
- $\emptyset \in \text{Rec}(\Sigma)$.
- $\Sigma \in \text{Rec}(\Sigma)$.
- $\forall L \in \text{Rec}(\Sigma), L^* \in \text{Rec}(\Sigma)$.
- $\forall L \in \text{Rec}(\Sigma), \bar{L} \in \text{Rec}(\Sigma)$.

Démonstration. On utilise, dans les graphes qui suivent, la notation $\xrightarrow{\Sigma}$ pour signifier que pour tout $a \in \Sigma$, l'arête \xrightarrow{a} est placée.

— Soit $a \in \Sigma$, le langage $\{a\}$ est reconnu par l'automate

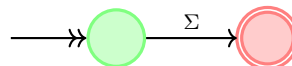


— Le langage $\{\varepsilon\}$ est reconnu par l'automate

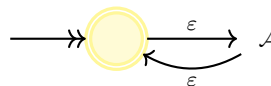


— L'automate à la fonction de transition "vide" reconnaît \emptyset

— L'automate suivant reconnaît Σ



— Soit $L \in \text{Rec}(\Sigma)$ reconnu par $\mathcal{A} := (\Sigma, Q, q_I, F, \delta)$. L'idée est de formaliser la construction intuitive :



- Soit $L \in \text{Rec}(\Sigma)$ reconnu par $\mathcal{A} := (\Sigma, Q, q_I, F, \delta)$. Un mot n'est pas dans L s'il n'est pas reconnu par \mathcal{A} si bien que l'automate $(\Sigma, Q, q_I, Q \setminus F, \delta)$ reconnaît \bar{L} .

■

Un curieux parallèle semble donc se dessiner entre les langages reconnus par les automates, ainsi que les langages réguliers. En effet, les deux vérifient les mêmes stabilités par opérations que les langages réguliers. C'est le sujet d'étude de la section suivante.

5.5 Théorème de Kleene

5.5.1 Le théorème

Théorème 5.31 (KLEENE). *Soit Σ un alphabet. On a $\text{Reg}(\Sigma) = \text{Rec}(\Sigma)$*

La preuve se déroule donc par double inclusion. Le sens indirecte fait l'objet du TP , mais il est pour le programme admis. On fait une preuve du sens réciproque dans la sous-section suivante.

5.5.2 Algorithme de Berry-Sethi

Définition 5.32. Un automate local est un automate déterministe $(\Sigma, Q, q_I, F, \delta)$ tel que lorsque toutes les transitions étiquetées par une lettre donnée vont vers le même état. Formellement, lorsque

$$\forall (q, q', a) \in Q \times Q \times \Sigma, \quad (\delta(q, a) \text{ existe}) \wedge (\delta(q', a) \text{ existe}) \implies \delta(q, a) = \delta(q', a)$$

Proposition 5.33. *Un langage L sur Σ est local si et seulement s'il est reconnu par un automate local.*

Démonstration. On procède par double implication :

$\boxed{\Rightarrow}$ Si L est local sur Σ engendré par (P, D, F, α) . On se donne \blacklozenge un objet quelconque qui n'est pas élément de Σ . Définissons pour tout $a \in P$:

$$\delta(\blacklozenge, a) := a,$$

et pour tout $(a, b) \in \Sigma \times (\Sigma \setminus P)$:

$$\delta(a, b) := b, \iff ab \in F.$$

Ensuite, si $\alpha = \top$ alors posons $F := D \sqcup \{\blacklozenge\}$ sinon $T := D$. Une simple vérification montre alors que l'automate $\mathcal{A} := (\Sigma, \Sigma \sqcup \{\blacklozenge\}, \blacklozenge, T, \delta)$ est local et reconnaît L .

$\boxed{\Leftarrow}$ Réciproquement, supposons que $\mathcal{A} := (\Sigma, Q, q_I, T, \delta)$ est local. Remarquons qu'en conséquence de cette hypothèse si $a \in \Sigma$, il existe au plus un état q_a tel que toutes les transitions étiquetées par a arrivent dans q_a . Posons alors :

- $P := \{a \in \Sigma : \delta(q_I, a) \text{ est défini}\}.$
- $D := \{a \in \Sigma : (q_a \text{ existe}) \wedge (q_a \in T)\}.$
- $F := \{ab \in \Sigma^2 : (q_a \text{ existe}) \wedge (\delta(q_a, b) \text{ est défini})\}.$
- Si $q_I \in T$, $\alpha := \top$ sinon $\alpha := \perp$.

On vérifie aisément que le langage local engendré par (P, D, F, α) , noté L , vérifie $L = L(\mathcal{A})$.

■

Étant donné e une expression régulière, l'algorithme de BERRY-SETHI consiste donc à réaliser les étapes suivantes :

- Calculer $\text{lin}(e)$
- Calculer le langage local de $\text{lin}(e)$
- Construire l'automate local qui reconnaît $\mathcal{L}(\text{lin}(e))$ (qui se nomme automate de Glushkov).
- Retirer les numéros (car $U(\mathcal{L}(\text{lin}(e))) = \mathcal{L}(e)$).

L'algorithme de BERRY-SETHI permet donc de construire un automate qui reconnaît le langage engendré par une expression régulière, on vient donc de montrer l'inclusion directe du Théorème de KLEENE.

Exercices

Exercice 5.1 (Exemples d'automates). 1. Trouver un automate fini déterministe qui reconnaît les entiers écrits en base 2 qui sont congrus à 1 modulo 3.

2. Donner un automate fini déterministe qui reconnaît l'ensemble des mots sur $\{a, b\}$ dont la i -ème lettre (i fixé) en partant de la fin est un a .

Exercice 5.2. Donner des automates reconnaissant les langages définis par les expressions régulières suivantes :

1. $(aab \mid aa \mid aab)^*$,
2. a^b ,
3. $\varepsilon \mid (a \mid aab)^* \mid a^*(aab)^*$.

Exercice 5.3 (Langages rationnels). Parmi les langages suivants, lesquels sont rationnels ?

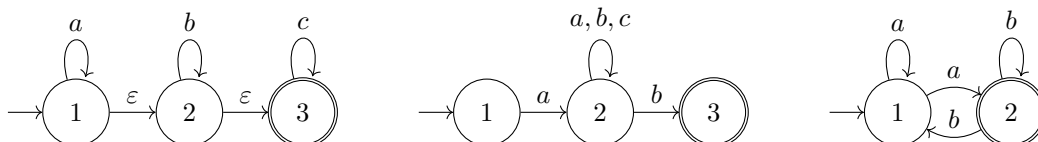
1. $\{a^{2n}, n \geq 0\}$.
2. $\{a^m b^n a^{m+n}, m \geq 0, n \geq 0\}$.
3. $\{w \mid |w|_a = |w|_b\}$.
4. $\{a^p \mid p \text{ est premier}\}$.
5. L'ensemble des mots qui n'ont pas trois a consécutifs.
6. L'ensemble des mots qui ont autant de a que de b .
7. L'ensemble des mots qui sont des palindromes sur $\{a, b\}$.
8. $\{a^i b^j, \text{pgcd}(i, j) = 1\}$.
9. $\{a^i b^j, i \geq j\}$.

Exercice 5.4 (Stabilité des langages reconnaissables). 1. Soit L un langage rationnel. Montrer que le langage $\{\bar{u}, u \in L\}$ est rationnel. (\bar{u} est le mot miroir de u .)

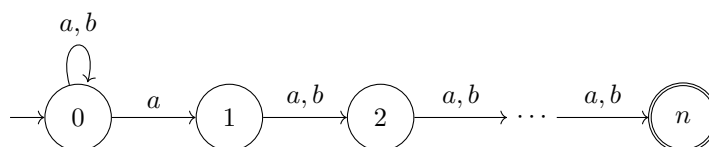
2. Soit L un langage rationnel. Montrer que le langage $\{x \in \Sigma^*, \exists y \in \Sigma^*, xy \in L\}$ est rationnel.

3. Soit L un langage rationnel. Montrer que le langage $\{x \in \Sigma^*, \exists y \in \Sigma^*, xy \in L, |x| = |y|\}$ est rationnel.

Exercice 5.5 (Déterminisation). Déterminiser les automates suivants.



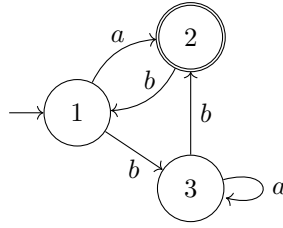
Exercice 5.6 (Déterminisation, encore). Combien d'état possède l'automate déterminisé obtenu à partir de l'automate suivant ?



Montrer qu'il existe pas d'automate déterministe complet équivalent et possédant strictement moins d'états.

Exercice 5.7 (Exemple de minimisation de Brzozowski). Écrire un automate non déterministe à 4 états, dont un seul est initial et un seul est final, qui reconnaît le langage engendré par $(b + ab)^* ba$. Renverser les transitions et déterminer l'automate miroir obtenu. Recommencer. Quel est le langage reconnu par le dernier automate ? Quelle autre propriété possède cet automate ?

Exercice 5.8 (Utilisation du lemme d'Arden). On considère l'automate :



On note, pour chaque état $q \in \{1, 2, 3\}$, L_q le langage reconnu par l'automate si l'état initial était q . Donner des égalités entre ces langages.

Ceci donne un système d'équations sur les langages. En utilisant le lemme d'Arden (voir la feuille sur les langages réguliers), résoudre ces équations. Donner une expression régulière qui engendre le langage reconnu par l'automate.

Exercice 5.9 (Automates accessibles). On considère dans cet exercice uniquement des automates finis déterministes. Deux automates sont *équivalents* s'ils reconnaissent le même langage.

On dit qu'un état $q \in Q$ de \mathcal{A} est *accessible* s'il existe un mot u tel que $\delta^*(i, u) = q$. L'automate est dit *accessible* si tous ses états le sont.

1. Rappeler la définition de δ^* .
2. Rappeler la définition du langage reconnu par \mathcal{A} , que l'on notera $L(\mathcal{A})$.
3. Montrer que tout automate fini déterministe est équivalent à un automate fini déterministe, complet et accessible.

Exercice 5.10 (Automates réguliers). On considèrera uniquement des automates complets accessibles. On dit qu'un automate \mathcal{A} est *régulier à droite* s'il vérifie la propriété suivante :

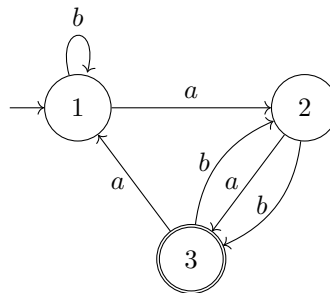
$$\forall u, v \in \Sigma^*, \delta^*(i, u) = \delta^*(i, v) \implies (\forall w \in \Sigma^*, \delta^*(i, uw) = \delta^*(i, vw))$$

Donner une condition suffisante pour que \mathcal{A} soit régulier à droite.

On dit qu'un automate \mathcal{A} est *régulier à gauche* s'il vérifie la propriété suivante :

$$\forall u, v \in \Sigma^*, \delta^*(i, u) = \delta^*(i, v) \implies (\forall w \in \Sigma^*, \delta^*(i, wu) = \delta^*(i, wv))$$

Soit S l'automate :



1. Montrer que S n'est pas régulier à gauche.

On veut montrer que tout automate complet déterministe est équivalent à un automate déterministe régulier à gauche. On va construire un automate dont l'ensemble des états est un sous-ensemble de Q^Q , l'ensemble de toutes les applications de Q dans lui-même.

Dans un but de simplification, on identifiera :

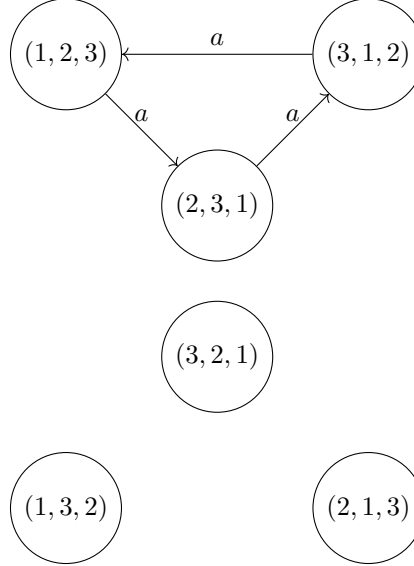
- l'ensemble Q avec $\llbracket 1, n \rrbracket$, où n est le cardinal de Q . On suppose que 1 est l'état initial ;
- chaque élément $r \in Q^Q$ au vecteur $(r(1), r(2), \dots, r(n))$. L'application identité est ainsi représentée par le vecteur $(1, 2, \dots, n)$.

On définit l'automate $\mathcal{A}_g = (\Sigma^*, Q^Q, id_Q, U, \eta)$. Son état initial est l'application identité. Une application r est acceptante ($r \in U$) si et seulement si $r(1) \in T$. Enfin, la fonction de transition η est définie par :

$$\forall r \in Q^Q, \forall a \in \Sigma, \forall k \in \llbracket 1, n \rrbracket, \eta(r, a)(k) = \delta(r(k), a)$$

En fait, on s'intéresse uniquement à la partie accessible de \mathcal{A}_g , que l'on note encore \mathcal{A}_g .

1. Compléter l'automate suivant pour donner la représentation graphique de S_g correspondant à l'automate S . Il manque 9 transitions et l'indication des états terminaux.



2. Vérifier que \mathcal{A}_g est déterministe.
3. Montrer que : $\forall r \in Q^Q, \forall u \in \Sigma^*, \forall k \in \llbracket 1, n \rrbracket, \eta^*(r, u)(k) = \delta^*(r(k), u)$.
4. Vérifier que \mathcal{A}_g est équivalent à \mathcal{A} .
5. Vérifier que \mathcal{A}_g est régulier à gauche.

Une application. Soit K un langage de Σ^* . Notons $\mathcal{R}_2(K) = \{f \in \Sigma^* \mid ff \in K\}$, l'ensemble des mots dont le carré est dans K .

Soit \mathcal{A} un automate fini déterministe. On note $V_2 = \{p \in Q \mid \exists f \in A^*, \delta^*(i, f) = p \text{ et } \delta^*(i, ff) \in T\}$ et $\mathcal{A}_{\mathcal{R}_2} = (\Sigma, Q, i, V_2, \delta)$ l'automate obtenu à partir de \mathcal{A} en prenant V_2 comme ensemble d'états terminaux.

1. Montrer que si \mathcal{A} est régulier à gauche, alors $L(\mathcal{A}_{\mathcal{R}_2}) = \mathcal{R}_2(L(\mathcal{A}))$.
2. En prenant $\mathcal{B} = S_g$, donner l'ensemble des états terminaux de $\mathcal{B}_{\mathcal{R}_2}$. On admettra, sans démonstration, qu'il suffit de considérer les mots dont le carré est de longueur inférieure ou égale à 6 pour calculer le V_2 correspondant.
3. Vérifier, avec l'automate S , que l'équation $L(\mathcal{A}_{\mathcal{R}_2}) = \mathcal{R}_2(L(\mathcal{A}))$ n'est pas toujours vérifiée pour un automate déterministe quelconque.
4. (Conclusion et généralisation.) Soit k un entier positif quelconque. Pour tout langage K de Σ^* , on note $\mathcal{R}_k(K) = \{f \in \Sigma^* \mid f^k \in K\}$. Montrer que si K est reconnu par un automate fini, alors $\mathcal{R}_k(K)$ est également reconnu par un automate fini.

Chapitre 6

Optimisation et approximation

Sommaire

6.0	Introduction	89
6.1	Briques élémentaires et algorithmes Gobelins	90
6.1.1	Définitions	90
6.1.2	L'algorithme glouton	90
6.1.3	Autour du voyageur de commerce	90
6.1.4	Le problème de la vache qui rit (ou du coupon collector)	92
6.2	Algorithmes probabilistes ou algorithmes Orcs	92
6.2.1	Algorithme de KARGER	93
6.2.2	Trions (avec le tri rapide)	94
6.2.3	Mélangeons (Algorithme de KNUTH FISHER-YATES)	96
6.3	Quelques difficultés concernant les probabilités en informatique	96

KARGER sépare, KARGER
contracte

F.Hatat

6.0 Introduction

On s'intéresse à des algorithmes qui à une entrée associent une valeur extrême de sortie d'une réponse à une entrée.

Exemples. Pour illustrer ce qui précède :

- Étant donnée une liste, donner son plus petit élément.
- Quelle est la taille d'un plus grand couplage dans un graphe ?
- Le problème du voyageur de commerce TSP qui renvoie, à partir d'un graphe pondéré, le plus petit poids d'un cycle hamiltonien du graphe.
- Soit G un graphe orienté pondéré. Quel est le plus grand flot de G d'un sommet s à un sommet t ? (MAXFLOW)
- Combien d'arêtes, au minimum, doit on retirer d'un graphe G pour qu'il perde sa connexité ? (MINCUT)

Remarque. On peut montrer que MINCUT se réduit à MAXFLOW et pire on ne sait toujours pas résoudre explicitement le premier. Toutefois, l'algorithme donnant MAXFLOW est affreux à réaliser en pratique.

Cependant, la complexité des algorithmes de résolutions pris comme exemple est assez variable, si bien que parfois l'on préfère calculer efficacement des approximations de solutions plutôt que des solutions exactes (c'est en particulier le cas pour les deux derniers problèmes de la liste). Certaines des notations concernant les problèmes qui seront traités en exemple proviennent des chapitres précédents.

6.1 Briques élémentaires et algorithmes Gobelins

6.1.1 Définitions

Définition 6.1 (problème de minimisation). Soit I un ensemble d'entrée et $S \subseteq I$ dit de solutions. On se donne deux fonctions

$$\text{Sol} : I \rightarrow \mathcal{P}(S), \quad \text{et} \quad w : S \rightarrow \mathbb{R}.$$

Un problème de minimisation est une fonction $\text{Opt} : I \rightarrow \mathbb{R}$ telle que

$$\forall i \in I, \quad \text{Opt}(i) = \min\{w(s)\}_{s \in \text{Sol}(i)}.$$

Remarque. Intuitivement, Sol est une fonction qui à chaque $i \in I$ associe toutes les solutions dans S . Aussi, on peut remplacer, dans la définition ci-dessus, \mathbb{R} par tout ensemble totalement ordonné (préférentiellement, des nombres). Enfin, en informatique, le minimum sera toujours bien défini.

Définition 6.2 (Problème de maximisation). On reprend les notations de la définition précédente. Lorsque $\text{Opt}(i) = \max\{w(s)\}_{s \in \text{Sol}(i)}$ on parle de problème de maximisation.

Définition 6.3 (Algorithme d'approximation de facteur λ). Si $\lambda \in \mathbb{R}$, $\text{Algo} : I \rightarrow \mathbb{R}$ est une λ approximation de Opt lorsque

- Si Opt est un problème de minimisation, $\forall i \in I, \text{Algo}(i) \leq \lambda \text{Opt}(i)$, ou encore $\frac{\text{Algo}}{\text{Opt}} \leq \lambda$
- Si Opt est un problème de maximisation, $\forall i \in I, \text{Opt}(i) \leq \lambda \text{Algo}(i)$, ou encore $\frac{\text{Opt}}{\text{Algo}} \leq \lambda$

tel que $\exists s \in \text{Sol}$ tel que $\text{Algo}(i) = w(s)$

Remarque. Dans la définition, on a en réalité $\lambda \geq 1$ car sinon, l'algorithme est meilleur que l'optimal ce qui n'a pas vraiment de sens.

Les trois exemples suivants sont bien sûr des illustrations qui ne figurent pas au programme.

6.1.2 L'algorithme glouton

Nous serions tenté d'imaginer un algorithme glouton fonctionnant de la sorte : on itère sur les arêtes, et pour chaque arête e , si elle n'est pas couverte, on ajoute ses extrémités à la couverture.

Proposition 6.4. *L'algorithme glouton proposé est une 2-approximation.*

Démonstration. Soit $G := (V, E)$ un graphe. On note m le nombre d'arêtes choisies par le glouton. La couverture donnée par glouton est de taille $2m$. On note par ailleurs \mathcal{G} l'algorithme glouton comme fonction mathématique. Soient e et e' deux arêtes choisies par glouton. Si e est choisie en premier, alors quand on examine e' , e' n'est pas encore couverte. Or les extrémités de e sont dans la couverture donc les extrémités de e et de e' sont disjointes. Dans une solution optimale, chacune des arêtes choisies par glouton doit être couverte. Pour chacune de ces m arêtes, l'optimal choisit au moins une des deux extrémités. Donc $\text{Opt} \geq m$. Donc $\text{Opt} \geq \frac{\mathcal{G}}{2}$ donc $2\text{Opt} \geq \mathcal{G}$ ■

6.1.3 Autour du voyageur de commerce

Imaginons qu'un marchand cherche à traverser une liste donnée de villes sur une carte. Entre ces villes, il existe des chemins qui permettent de passer de l'une à l'autre. Une question peut venir pour optimiser les coûts de ce voyageur, quel est le plus court circuit passant par toutes les villes, une et une seule fois ? Ce problème est celui du voyageur de commerce. Formellement

Définition 6.5 (Problème du voyageur de commerce). On désigne par TSP le problème de décision prenant en entrée :

- un graphe pondéré complet non orienté G ,

— une borne B ,

décidant s'il existe un cycle hamiltonien de poids plus petit que B (que l'on appelle *tournee*).

TSP est un problème **NP**-complet, par conséquent, on ne connaît à ce jour de moyen pour le résoudre en temps polynomial. On pourrait donc chercher à approximer les solutions de ce problème. Néanmoins, quelques difficultés apparaissent.

Théorème 6.6 (Inapproximabilité du voyageur de commerce). *Si $\mathbf{P} \subsetneq \mathbf{NP}$, il n'existe aucune approximation polynomiale du voyageur de commerce.*

Démonstration. Raisonnons par l'absurde. S'il existait une λ approximation de TSP notée A , alors on sait résoudre cycle hamiltonien. En effet, soit $G := (V, E)$ un graphe et considérons son complété $G' := (V, \mathcal{P}_2(V))$ auquel on ajoute des poids aux arêtes :

$$\forall e \in \mathcal{P}_2(V), \quad w(e) := \begin{cases} 1 & \text{si } e \in E \\ \lambda|V| + 1 & \text{sinon} \end{cases}$$

Observons que

- S'il existe un cycle hamiltonien dans G , alors G' a une tournée de poids $|V|$ de sorte que A renvoie une valeur comprise entre $|V|$ et $\lambda|V|$ (car c'est une λ -approximation).
- Si G' n'a pas de cycle hamiltonien alors toute tournée dans G' passe par une arête $e \notin E$ donc est de poids au minimum $\lambda|V| + 1$ (le poids d'une arête $e \notin E$) ? Donc A donne une tournée de poids $\lambda|V| + 1$.

Par conséquent G a un cycle hamiltonien si et seulement si $A(G') \leq \lambda|V|$. Mais puisque l'étape de complétion de G est polynomiale en la taille de G , on vient donc d'exhiber une réduction polynomiale de chemin hamiltonien vers le voyageur de commerce d'où le résultat. ■

Remarque. Si l'on considère la probabilité qu'une formule logique soit satisfiable en fonction du rapport du nombre de clauses n sur le nombre de littéraux p , cette fonction est croissante (tracer la courbe?).

On peut considérer des sous classes, on définit un gap, le segment sur lequel la dérivée de notre dite fonction est la plus forte. Les problèmes se trouvant dans ce gap sont inapproximables.

Le gap de la réduction de VERTEXCOVER à SAT est un gap immense, il est donc quasiment toujours inapproximable.

Nous allons étudier une variante. On suppose désormais que les sommets sont des points du plan et que la distance entre les sommets est la distance euclidienne, les chemins étant les segments reliant ces points.

Théorème 6.7 (Cas d'approximabilité du voyageur de commerce). *Il existe une 2-approximation de TSP euclidien*

On commence par calculer un arbre couvrant de poids minimal. On choisit une racine arbitraire. On lance un parcours en profondeur. On note les sommets dans l'ordre dans lequel on les rencontre (à chaque passage).

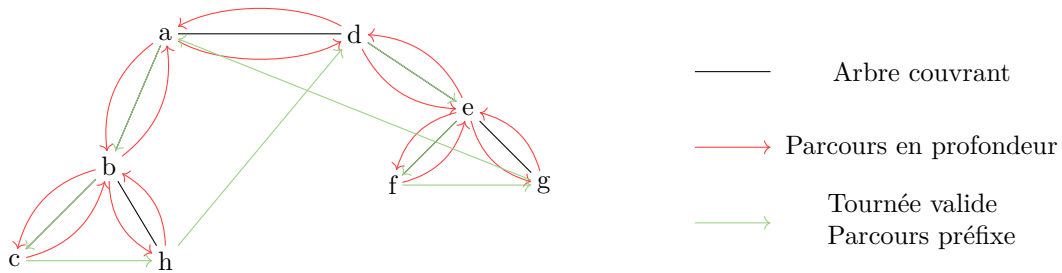


FIGURE 6.1 – Algorithme d'approximation du voyageur de commerce

Exemple. Cela donne : $abcbhbadefegeda$. Pour rendre le cycle hamiltonien, à chaque fois qu'on passe par un sommet déjà visité, on l'ignore et on passe au suivant. On obtiendrait donc $abchdfga$. Montrons que ce tour est à un facteur au plus 2 de l'optimal.

Démonstration. Si on note \mathcal{T} l'arbre couvrant dont on est parti, on note c la fonction qui à un ensemble d'arêtes donne le poids total. On remarque que $c(\mathcal{T}) \leq \text{Opt}$, où Opt est le poids de la tournée optimale du voyageur de commerce. En effet, si on prend une tournée du TSP et si on lui retire une arête, c'est un arbre couvrant. Ainsi une solution de vertex cover est un arbre couvrant particulier. Or \mathcal{T} est l'arbre couvrant de poids minimal. ■

Remarque. Si on note w le premier cycle (non hamiltonien) qu'on a construit, on a

- $c(w) = 2c(\mathcal{T})$
- On note β le cycle hamiltonien qu'on a construit. On a $c(\beta) \leq c(w)$ d'après l'inégalité triangulaire, puisque si w est formée des arêtes $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \xrightarrow{e_3} \dots$ alors β prend soit des arêtes de w , soit des raccourcis en sautant quelques sommets. Sur un saut, l'inégalité triangulaire

$$d(e_i, e_j) \leq \sum_{k=i}^{j-1} d(e_k, e_{k+1})$$

d'où l'on tire finalement que

$$c(\beta) \leq c(w) = 2c(\mathcal{T}) \leq 2\text{Opt}.$$

On a bien une 2-Approximation

Remarque (Pour la culture). On a vu des exemples d'approximation polynomiales à un facteur constant. Il existe des schémas d'approximation polynomiaux pour certains problèmes. Ce sont des classes d'algorithmes qui donnent à chaque $\varepsilon > 0$, donnent un algorithme d'approximation polynomial à un facteur $1 + \varepsilon$. Cependant on a toujours pas $\mathbf{P} = \mathbf{NP}$, en effet, le degré du polynôme dépend de $\frac{1}{\varepsilon}$.

6.1.4 Le problème de la vache qui rit (ou du coupon collector)

L'exemple qui suit n'est pas un problème de décision mais un problème de probabilité que l'on croise dans des situations assez diverses. Soit X le nombre de boîtes à acheter pour avoir une collection de n vignettes. Que vaut $\mathbb{E}(X)$?

On considère X_i le nombre de boîtes à acheter pour passer de i vignettes à $i + 1$ vignettes. On a :

- $X_0 = 1 \hookrightarrow \mathcal{G}(1)$
- $X_1 \hookrightarrow \mathcal{G}(\frac{n-1}{n})$
- $X_k \hookrightarrow \mathcal{G}(\frac{n-k}{n})$

De plus, on a $X = \sum_{k=0}^{n-1} X_k$, si bien que par linéarité de l'espérance

$$\mathbb{E}(X) = \sum_{k=0}^{n-1} \mathbb{E}(X_k) = \sum_{k=0}^{n-1} \frac{n}{n-k} = n \sum_{k=0}^{n-1} \frac{1}{j} \sim n \ln(n)$$

6.2 Algorithmes probabilistes ou algorithmes Orcs

L'apparition des probabilités concernant les algorithmes peut intervenir à deux niveaux : la terminaison de celui-ci et à plus forte raison la correction de celui-ci. Donnons quelques définitions.

Définition 6.8 (Algorithme de Las Vegas). Un algorithme est qualifié de *Las Vegas* lorsque le temps de calcul de celui-ci est une variable aléatoire mais dès lors qu'il termine, il renvoie une réponse correcte.

Exemple. On cherche à simuler des tirages sans remises. L'idée intuitive c'est de piocher tant que l'on peut encore dans une urne en stockant au fur et à mesure les nombres tirés. Ceci donne l'algorithme de la page suivante :

où dans le pseudo-code ci-dessus $\text{TIRER}(a, b)$ tire un nombre suivant une loi de probabilité uniforme sur $\llbracket a, b \rrbracket$.

Par ailleurs, soit A est l'évènement "faire une infinité de tirages de boules déjà vues" on sait que $\mathbb{P}(A) = 0$. L'algorithme est un Las Vegas qui termine avec une probabilité 1.

Remarque. Le tri rapide est un Las Vegas. Plus largement, tout algorithme qui possède un pire cas différent d'un meilleur cas.

Définition 6.9 (Algorithme de Monte-Carlo). Un algorithme est qualifié de *Monte-Carlo* lorsqu'il s'exécute en temps fini mais qui ne donne la bonne réponse que de façon probabiliste.

Les informaticiens aiment les casinos visiblement.

Les trois exemples suivants illustrant le propos sont bien sûr hors programme.

Algorithme 6.1 Simuler un tirage sans remise de n entiers dans un ensemble $\llbracket 0, n-1 \rrbracket$

$\text{vu} \leftarrow \text{Tableau à } n \text{ cases initialisées à } \perp$

Pour i de 1 à n , **faire** :

$k \leftarrow \text{TIRER}(0, n-1)$

 Tirer k entre $\llbracket 0, n-1 \rrbracket$

Tant que $\text{vu}(k)$, **faire** :

 Tirer à nouveau k

Fin tant que

$\text{vus}[k] \leftarrow \top$

Fin pour

6.2.1 Algorithme de Karger

Définition 6.10. (MINCUT : KARGER) On considère un graphe G non orienté et on veut déterminer le cardinal du plus petit ensemble d'arêtes C tel que $G \setminus C$ n'est pas connexe (cela s'appelle *une coupe*). L'algorithme de KARGER raisonne de façon probabiliste.

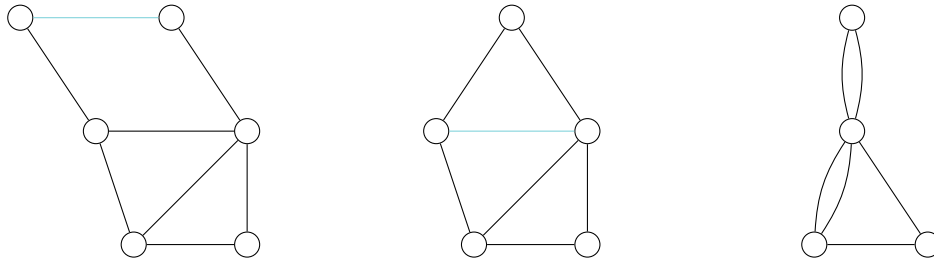


FIGURE 6.2 – 2 transformations successives par l'algorithme de KARGER

Au hasard il choisit une arête, il la supprime, il fusionne ses extrémités et recommence jusqu'à ce qu'il ne reste plus que 2 sommets. Les arêtes entre ces deux sommets forment une coupe.

En vu d'obtenir une correction probabiliste de l'algorithme de KARGER, commençons par un lemme.

Lemme 6.11. Si C est une coupe minimale d'un graphe $G = (V, E)$ alors $2\#E \geq \#V\#C$.

Démonstration. Soit $v \in V$, comme C est une coupe minimale alors $\deg(v) \geq \#C$, sans quoi en retirant une arête des voisins de v on obtiendrait une coupe de cardinal plus petit que le $\#C$ en contradiction avec l'hypothèse faite. Par suite, en sommant

$$\sum_{v \in V} \deg(v) \geq \#V\#C$$

d'où, avec le lemme des poignées de mains $2\#E \geq \#V\#C$. ■

Remarque. Cette propriété reste vraie à tout instant lors de l'exécution de l'algorithme de KARGER y compris en présence d'arêtes parallèles.

Théorème 6.12. Si le graphe a n sommets considérons l'évènement "KARGER donne une coupe minimale". On a $\mathbb{P}(A) \geq \frac{2}{n(n-1)}$.

Démonstration. Soit E_i l'évènement "à la i -ème étape, on ne choisit pas une arête de C ". Posons

$$\forall k \in \mathbb{N}, F_k := \bigcap_{i=1}^k E_i,$$

et observons que KARGER trouve la coupe C lorsque l'évènement F_{n-2} est réalisé, on cherche donc à estimer la probabilité $\mathbb{P}(F_{n-2})$. Au début de l'étape i de l'algorithme de KARGER, le graphe contient

- une coupe de taille $\#C$
- $n - i + 1$ sommets

Si c désigne le nombre d'arêtes restantes alors d'après le lemme ci-dessus et la remarque l'accompagnant, $2c \geq \#C(n - i + 1)$ donc

$$P_{E_1 \cap \dots \cap E_{i-1}}(\overline{E_i}) \leq \frac{\#C}{\frac{k(n-i+1)}{2}} \leq \frac{2}{n-i+1}$$

donc en passant par l'évènement complémentaire

$$P_{E_1 \cap \dots \cap E_{i-1}}(E_i) \geq 1 - \frac{2}{n-i+1}$$

Par conséquent, la formule des probabilités composées et ce qui précède permettent d'obtenir :

$$\mathbb{P}(F_{n-2}) \geq \prod_{k=1}^{n-2} \left(1 - \frac{2}{n-k+1}\right) = \prod_{k=1}^{n-2} \frac{n-k-1}{n-k+1} = \frac{\prod_{j=1}^{n-2} j}{\prod_{j=3}^n j} = \frac{2}{n(n-1)}$$

c'est l'inégalité désirée. ■

Remarque. L'algorithme de KARGER s'exécute en temps fixé mais ne renvoie la bonne réponse qu'avec une certaine probabilité, c'est donc un algorithme Monte-Carlo. Par conséquent, sur le problème de décision : "étant donné G et $\#C$, existe-t-il une coupe de G de taille $\leq \#C$?" Quand KARGER répond vrai : il a raison. Quand KARGER répond faux : c'est faux avec une certaine probabilité (mais KARGER répond faux sur toutes les fois où c'est réellement faux).

Il est naturelle de penser que la borne du théorème que l'on vient de démontrer n'est pas si bonne que cela. En revanche, rien ne nous empêche de relancer plusieurs fois l'algorithme de KARGER de sorte à réduire la probabilité de ne pas tomber sur *une* coupe minimale. Si on lance KARGER t fois indépendamment, notons E_t l'évènement "KARGER trouve une coupe minimale en t exécutions". On a donc si les R_i sont les évènements "KARGER ne renvoie pas une coupe minimale au i -ème essai" (qui sont indépendants),

$$\mathbb{P}(\overline{E_t}) = \mathbb{P}\left(\bigcap_{i=1}^t R_i\right) = \prod_{i=1}^t \mathbb{P}(R_i) \leq \left(1 - \frac{2}{n(n-1)}\right)^t,$$

par indépendance des exécutions. Ainsi, en prenant le complémentaire

$$\mathbb{P}(E_t) \geq 1 - \left(1 - \frac{2}{n(n-1)}\right)^t.$$

En particulier, si l'on choisit $t = n^2$ il vient classiquement que le membre de droite tend vers $1 - 1/e^2$ lorsque n tend vers $+\infty$ donc en particulier pour t assez grand $\mathbb{P}(E_t) \geq 1 - 1/e$. On améliore alors d'autant plus la convergence qu'on applique de multiples fois KARGER ce qui a pour écueil d'augmenter la complexité (on reste polynomial cependant).

6.2.2 Trions (avec le tri rapide)

Une application notable des algorithmes probabilistes est celle des algorithmes de tri dont voici une définition.

Définition 6.13 (Algorithme de tri). C'est un algorithme qui fait des comparaisons pour trouver une permutation.

On peut caractériser l'exécution d'un algorithme de tri par la suite des comparaisons qu'il effectue. En effet, un algorithme de tri déterministe effectue toujours la même $(k+1)$ -ième comparaison sur toutes les entrées qui donnent les mêmes résultats aux k -ième premières comparaisons. Fort de cette remarque, l'algorithme de tri peut être décrit par un arbre binaire où chaque noeud est une comparaison entre deux éléments telle que le fils gauche est la comparaison suivante si la racine donne faux, et le fils droit si la racine donne vrai. Pour mieux comprendre cette idée prenons l'exemple du tri rapide

Exemple (Arbre du tri rapide). On se restreint à un tri à trois éléments, sinon l'arbre devient assez gros :

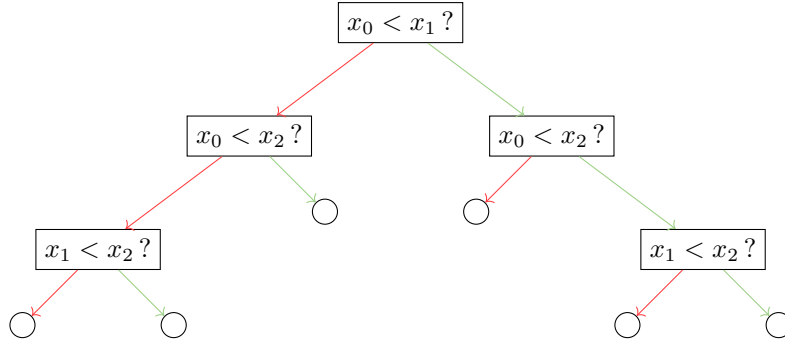


FIGURE 6.3 – Arbre d'un tri à 3 éléments

On attribue à chaque noeud l'ensemble des permutations compatibles avec les comparaisons effectuées. La racine porte les $n!$ permutations. L'algorithme est correct donc les feuilles sont étiquetées par une permutation seule. De manière plus générale, un algorithme de tri est pareil à un arbre binaire qui possède au moins $n!$ feuilles (au moins parce que rien n'empêche d'avoir un tri stupide). Par suite, les caractéristiques de l'arbre d'un tri rendent compte des propriétés même de l'algorithme de tri. À titre d'exemple, on remarque que la complexité temporelle d'un algorithme pour une taille d'entrée donnée se retrouve sur la hauteur de celui-ci.

Théorème 6.14. *Soit $(E, <)$ un ensemble totalement ordonné. Sans hypothèses additionnelles, tout algorithme de tri a une complexité d'au moins $O(n \log(n))$.*

Démonstration. On montre aisément que tout arbre binaire de hauteur h possède au plus 2^h feuilles (par récurrence ou induction structurelle). Soit \mathcal{A} un algorithme de tri sur $(E, <)$. Pour tout entrée de taille $n \in \mathbb{N}$, l'arbre de \mathcal{A} a au moins $n!$ feuilles. Au vu de ce qui précède, l'arbre de \mathcal{A} a donc une hauteur d'au moins $\log(n!)$ feuilles si bien que d'après la formule de Stirling et par composition des équivalents au logarithme $\ln(n!) \sim \log(n^n e^{-n} \sqrt{2\pi n}) = O(n \log(n))$ ce qu'on voulait montrer. ■

Un tri, c'est (aussi) un algorithme qui prend en entrée une permutation σ et qui calcule σ^{-1} . On suppose que σ est tirée aléatoirement et uniformément.

Proposition 6.15. *Soit X la variable aléatoire donnant le temps de calcul exacte pour une entrée de taille $n \in \mathbb{N}$ à fournir au tri rapide. On a $\mathbb{E}(X) = O(n \log(n))$.*

Démonstration. En sortie, on récupère la liste $[\sigma(0), \sigma(1), \dots, \sigma(n-1)]$. $\sigma(i)$ étant l'entier qui se trouve en position i après tri. On note $X_{i,j}$ la variable aléatoire qui vaut 1 si l'algorithme a comparé $\sigma(i)$ et $\sigma(j)$, et 0 sinon. $X_{i,j}$ est une épreuve de Bernoulli. En effet, à quelle condition a-t-on comparé $\sigma(i)$ et $\sigma(j)$ durant le tri rapide ? On ne fait que des comparaisons avec le pivot. $\dots, \sigma(i), \sigma(i+1), \dots, \sigma(j-1), \sigma(j)$. Si $\sigma(i)$ est le premier à être visité, alors $X_{i,j} = 1$. De même si $\sigma(j)$ est le premier à servir de pivot. Si le premier à être choisi comme pivot, c'est l'un des $\sigma(i+1), \dots, \sigma(j-1)$. Alors $X_{i,j} = 0$. Donc,

$$\mathbb{P}(X_{i,j} = 1) = \frac{2}{j-i+1},$$

c'est ce qu'on avait prétendu. Par suite remarquons que

$$X = \sum_{1 \leq i < j \leq n} X_{i,j}.$$

La linéarité de l'espérance donne alors :

$$\mathbb{E}(X) = \sum_{1 \leq i < j \leq n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k},$$

si bien que,

$$\mathbb{E}(X) \leq 2 \sum_{i=1}^n \sum_{k=1}^n \frac{1}{k} = O(n \log n),$$

c'est le résultat désiré. ■

Remarque. On retrouve donc ainsi l'idée que la complexité du tri rapide est en moyenne de $O(n \log(n))$.

Remarque. L'intérêt du mélange est, sur le plan pratique, totalement inutile puisque l'on peut raisonnablement estimer qu'une liste donnée à un algorithme de tri est déjà tirée aléatoirement (et uniformément) parmi toutes les listes possibles (intuitivement, ça ne fait pas grand sens autrement). L'utilisation d'un tel mélange est ici théorique : se placer dans le cadre d'application de la proposition qui précède.

Algorithme 6.2 RandQuickSort

Entrée : une liste l

Sortie : tri la liste donnée

fonction RANDQUICKSORT(L) :

MELANGE(l)

QUICKSORT(l)

Fin fonction

6.2.3 Mélangeons (Algorithme de Knuth Fisher-Yates)

faire tableaux ici

Algorithme 6.3 Mélange de KNUTH FISHER-YATES

Entrée : un tableau t de taille n

Sortie : une permutation de t uniformément tiré

fonction KFY(t) :

Pour i de $n-1$ à 0 , **faire** :

$j \leftarrow \text{TIRER}(0, i)$

échanger $t[i]$ et $t[j]$

Fin pour

Fin fonction

Proposition 6.16. *L'algorithme KFY tire uniformément une permutation du tableau en entrée.*

Remarque. On peut reformuler l'énoncé de la proposition comme suit : soit \mathbf{t} un tableau, si Σ est la variable aléatoire donnant la permutation tirée, alors $\Sigma \hookrightarrow \mathcal{U}(\mathfrak{S}(\mathbf{t}))$. On a par conséquent :

$$\forall \sigma \in \mathfrak{S}(\mathbf{t}), \quad \mathbb{P}(\Sigma = \sigma) = \frac{1}{|\mathbf{t}|!}.$$

Démonstration. Après la première étape, l'algorithme recommence sur un tableau de taille $n-1$. On prouve par récurrence que le tirage est uniforme :

Soit E l'évènement "On place le dernier élément V comme σ ". Remarquons que (E, \overline{E}) est un système complet d'évènement, si bien que la formule des probabilités totales donne

$$\mathbb{P}(\Sigma = \sigma) = \mathbb{P}_E(\Sigma = \sigma)\mathbb{P}(E) + \underbrace{\mathbb{P}_{\overline{E}}(\Sigma = \sigma)\mathbb{P}(\overline{E})}_{=0} = \mathbb{P}_E(\Sigma = \sigma) \times \frac{1}{n} = \frac{1}{(n-1)!} \times \frac{1}{n},$$

où la dernière égalité vient de l'hypothèse de récurrence. ■

6.3 Quelques difficultés concernant les probabilités en informatiques

Ajouter les explications sur les problèmes relatifs aux flottants.

Exemple (Simulation d'une loi de POISSON en C). On peut penser au code suivant :

```
int poisson (float lambda){
    int k = 0;
    X = rand dans [0, 1[;
    float somme = exp(- lambda) * 1 / 1;
    while(somme < x){
        somme += exp(- lambda) lambda^k / k!;
    }
    return k;
}
```


Chapitre 7

Logique

Sommaire

7.0	Banalités	97
7.1	Règles de déduction naturelles	97
7.2	Correction de la déduction naturelle	100
7.3	Dans la vie il y a 42 implications, ou comment se tordre la tête	100
7.4	Quantificateurs (1 ^{er} ordre)	101
	Exercices	102

Le tiers exclu, c'est le diable

F.Hatat

7.0 Banalités

Définition 7.1. Comme l'an passé, l'ensemble des formules \mathcal{F} est défini comme le plus petit ensemble construit inductivement par

$$\mathcal{F} = \mathcal{V} \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \mathcal{F} \rightarrow \mathcal{F} \mid \top \mid \perp.$$

Une formule ne veut donc *a priori* rien dire. Pour lui donner un sens on considère une valuation $\rho : \mathcal{V} \rightarrow \mathbb{B}$ sur laquelle on peut définir l'interprétation d'une formule $f \in \mathcal{F}$, $\llbracket f \rrbracket_\rho$ récursivement.

Par suite, nous avons également introduit les interprétation $\rho \models f$, c'est à dire $\llbracket f \rrbracket_\rho = 1$ et plus généralement

$$\Gamma_1, \dots, \Gamma_n \models f \stackrel{\text{(déf)}}{\iff} \forall \rho : \mathcal{V} \rightarrow \mathbb{B}, \quad (\forall i \in [1, n], \quad \llbracket \Gamma_i \rrbracket_\rho = 1 \implies \llbracket f \rrbracket_\rho = 1)$$

Attention, une interprétation ne fournit pas de preuve. Si on veut prouver $a \wedge a \rightarrow a$, cela donne : Supposons $a \wedge a$. Donc a , à cause de la gauche de $a \wedge a$. Voici une autre preuve : Supposons $a \wedge a$. Donc a grâce à la droite de $a \wedge a$. Pour prouver $a \wedge a \rightarrow a \wedge a$, on peut fournir 5 preuves différentes.

7.1 Règles de déduction naturelles

Définition 7.2 (Séquent). Un séquent est un couple (Γ, A) où Γ est une liste de formules et A est une formule. On le note $\Gamma \vdash A$. Γ est appelé *contexte*.

Définition 7.3 (Règle de déduction naturelle). Une règle de déduction est composée de deux séquents séparés par une bar de fraction. Le séquent du haut constitue les prémisses et celui du bas la conclusion. On indique à côté de la barre le nom de la règle.

On définit dans la suite quelques règles de déductions naturelles usuelles.

Définition 7.4 (L'axiome). L'axiome est une règle qui n'a pas de prémisses, qui permet de "finir" une branche de l'arbre de preuve :

$$\frac{}{\Gamma, A \vdash A} \text{ (ax)}$$

Définition 7.5 (Règles pour la \rightarrow).

Introduction de la \rightarrow : Élimination de la \rightarrow :

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \text{ (Intro} \rightarrow \text{)} \quad \frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \text{ (Elim} \rightarrow \text{)}$$

Définition 7.6 (Règles pour le \wedge).

Introduction du \wedge : Élimination du \wedge à gauche : Élimination du \wedge à droite :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \text{ (Intro } \wedge \text{)} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \text{ (Elim G } \wedge \text{)} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \text{ (Elim D } \wedge \text{)}$$

Définition 7.7 (Règles pour le \vee).

Introduction du \vee à gauche : Introduction du \vee à droite :

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \text{ (Intro G } \vee \text{)} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \text{ (Intro D } \vee \text{)}$$

Élimination du \vee :

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \text{ (Elim } \vee \text{)}$$

Définition 7.8 (Règles pour \top et \perp).

Introduction du \top : *Ex falso quodlibet sequitur*, ou efqs :

$$\frac{}{\Gamma \vdash \top} \text{ (Intro } \top \text{)} \quad \frac{\Gamma \vdash \perp}{\Gamma \vdash A} \text{ (efqs / } \perp \text{e)}$$

Définition 7.9 (Autres opérations). On introduit des notations purement syntaxiques, qui ne disposent aucunement de règles de déduction naturelle, mais allègent les notations :

$$\neg A := A \rightarrow \perp \quad A \leftrightarrow B := A \rightarrow B \wedge B \rightarrow A$$

Cet ensemble de règles, ce sont les règles de la logique intuitionniste .

Exemple. Montrer que $a \rightarrow b \rightarrow a \rightarrow a$.

$$\frac{\frac{\frac{\frac{}{a, b, a \vdash a} \text{ (ax)}}{a, b \vdash a \rightarrow a} (\rightarrow \text{ i})}{a \vdash b \rightarrow a \rightarrow a} (\rightarrow \text{ i})}{\vdash a \rightarrow b \rightarrow a \rightarrow a} (\rightarrow \text{ i})$$

Remarque. C'est ce que l'on appelle un arbre de preuve où les noeuds sont les règles de déduction et les enfants sont les étapes qui viennent logiquement après.

Exemple. Montrons que $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$. Nous avons l'arbre de preuve suivant en posant $\Gamma = a \rightarrow b \rightarrow c, b, a$:

$$\frac{\frac{\frac{\frac{}{\Gamma \vdash a \rightarrow b \rightarrow c} \text{ (ax)}}{\Gamma \vdash b \rightarrow c} \text{ (Elim} \rightarrow \text{)}}{\frac{\frac{\frac{}{\Gamma \vdash a} \text{ (ax)}}{\Gamma \vdash b} \text{ (Elim} \rightarrow \text{)}}{\frac{\frac{\frac{\frac{}{a \rightarrow b \rightarrow c, b, a \vdash c} \text{ (ax)}}{a \rightarrow b \rightarrow c, b \vdash a \rightarrow c} (\rightarrow \text{ i})}{a \rightarrow b \rightarrow c, b \vdash a \rightarrow c} (\rightarrow \text{ i})}{a \rightarrow b \rightarrow c \vdash b \rightarrow a \rightarrow c} (\rightarrow \text{ i})}{\vdash (a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c} (\rightarrow \text{ i})$$

Exemple. Montrons la commutativité du \vee :

$$\frac{\frac{\overline{a \vee b, b \vdash b}^{(ax)}}{a \vee b, b \vdash b \vee a}^{(\vee i)} \quad \frac{\frac{\overline{a \vee b, a \vdash a}^{(ax)}}{a \vee b, a \vdash b \vee a}^{(\vee i)} \quad \frac{\overline{a \vee b \vdash a \vee b}^{(ax)}}{a \vee b \vdash b \vee a}^{(Elim \vee)}}{\vdash a \vee b \rightarrow b \vee a}^{(\rightarrow i)}$$

Exemple. Un exemple de la règle *Ex falso quodlibet sequitur* :

$$\frac{\frac{\overline{a \wedge \neg a \vdash a \wedge \neg a}^{(ax)}}{a \wedge \neg a \vdash a}^{(Elim \wedge)} \quad \frac{\frac{\overline{a \wedge \neg a \vdash a \wedge \neg a}^{(ax)}}{a \wedge \neg a \vdash \neg a}^{(Elim \wedge)}}{a \wedge \neg a \vdash \perp}^{(BOUM! \text{ efqs})} \quad \frac{\overline{a \wedge \neg a \vdash \perp}}{a \wedge \neg a \vdash b}^{(Intro \rightarrow)} \quad \frac{\overline{a \wedge \neg a \vdash b}}{\vdash a \wedge \neg a \rightarrow b}^{(Intro \rightarrow)}$$

Certaines formules restent indémontrables avec nos outils actuels. Forgeons nous un nouvel accessoire :

Définition 7.10 (Tiers-exclu). On définit le tiers-exclu comme $\overline{\Gamma \vdash A \vee \neg A}^{(1/3)}$

Après avoir ajouté cette règle, on obtient la *logique classique*. On peut donc démontrer une formule qui, jusqu'alors, posait problème : $\neg\neg p \rightarrow p$.

Exemple ($\neg\neg p \rightarrow p$). En effet,

$$\frac{\frac{\overline{\neg\neg p \vdash p \vee \neg p}^{(1/3)} \quad \frac{\overline{\neg\neg p, p \vdash p}^{(ax)}}{\neg\neg p \vdash p}^{(\rightarrow i)} \quad \frac{\frac{\overline{\neg\neg p, \neg p \vdash \neg p}^{(ax)}}{\neg\neg p, \neg p \vdash \perp}^{(Efqs)} \quad \frac{\overline{\neg\neg p, \neg p \vdash \neg p}^{(ax)}}{\neg\neg p, \neg p \vdash p}^{(Elim \rightarrow)}}{\neg\neg p \vdash p}^{(Elim \vee)}$$

Exemple (Formule de Peirce). Montrons que $((a \rightarrow b) \rightarrow a) \rightarrow a$

$$\frac{T \quad \frac{\overline{(a \rightarrow b) \rightarrow a \vdash a \vee \neg a}^{(1/3)} \quad \frac{\overline{(a \rightarrow b) \rightarrow a, a \vdash a}^{(ax)}}{(a \rightarrow b) \rightarrow a, a \vdash a}^{(\vee e)}}{(a \rightarrow b) \rightarrow a \vdash a}^{(\rightarrow i)} \quad \frac{\overline{(a \rightarrow b) \rightarrow a \vdash a}}{\vdash ((a \rightarrow b) \rightarrow a) \rightarrow a}^{(\rightarrow i)}$$

où T est l'arbre de preuve suivant :

$$\frac{\overline{(a \rightarrow b) \rightarrow a, \neg a \vdash (a \rightarrow b) \rightarrow a}^{(ax)}}{(a \rightarrow b) \rightarrow a, \neg a \vdash a}^{(\rightarrow e)} \quad T'$$

et T' l'arbre suivant :

$$\frac{\frac{\overline{(a \rightarrow b) \rightarrow a, a, \neg a, a \vdash \neg a}^{(ax)}}{(a \rightarrow b) \rightarrow a, a, \neg a, a \vdash \perp}^{(efqs)} \quad \frac{\overline{(a \rightarrow b) \rightarrow a, a, \neg a, a \vdash a}^{(ax)}}{(a \rightarrow b) \rightarrow a, a, \neg a, a \vdash b}^{(\rightarrow i)}}{(a \rightarrow b) \rightarrow a, \neg a \vdash (a \rightarrow b)}^{(\rightarrow i)}$$

Remarque. On se rend compte que certaines formules se démontrent ne peuvent se démontrer qu'à l'aide du tiers exclu. On a donc $LJ \subset LK$. L'inclusion est d'ailleurs stricte, mais cette preuve est plus complexe.

7.2 Correction de la déduction naturelle

Théorème 7.11. *Si $\Gamma \vdash A$, alors $\Gamma \models A$*

Démonstration. On démontre le résultat par induction structurale sur l'arbre de preuve de $\Gamma \vdash A$.

- Si $\Gamma \vdash A$ est obtenu par application de la règle axiome, alors $A \in \Gamma$. Ainsi, soit ρ une valuation telle que $\rho \models \Gamma$. Par conséquent, $\forall \varphi \in \Gamma$, $\llbracket \varphi \rrbracket_\rho = \top$, or $A \in \Gamma$ si bien que $\llbracket A \rrbracket_\rho = \top$ d'où $\Gamma \models A$.
- Si $\Gamma \vdash A$ est obtenu par application de la règle (Intro \rightarrow) : $\frac{\dots}{\Gamma, B \vdash C} (\dots)$ $\frac{\Gamma, B \vdash C}{\Gamma \vdash A} (\rightarrow i)$ alors $A = B \rightarrow C$. Par hypothèse d'induction, $\Gamma, B \models C$. Soit ρ tel que $\rho \models \Gamma$.
 - Si $\llbracket B \rrbracket_\rho = 0$ alors par définition de $\llbracket B \rightarrow C \rrbracket_\rho$, on a $\llbracket B \rightarrow C \rrbracket_\rho = \llbracket A \rrbracket_\rho = 1$
 - Si $\llbracket B \rrbracket_\rho = 1$, or $\rho \models \Gamma$ donc $\rho \models B$ d'après l'hypothèse d'induction, $\llbracket C \rrbracket_\rho = 1$ donc $\llbracket A \rrbracket_\rho = 1$
- Etc...

■

Corollaire 7.12. *On a $\not\models \perp$ donc $\not\vdash \perp$.*

À titre culturel, on a le résultat suivant :

Théorème 7.13 (Complétude (HP)). *Si $\Gamma \models A$ alors $\Gamma \vdash A$ (en logique classique)*

Démonstration. Une preuve particulièrement élégante ne peut malheureusement tenir dans ce texte (mais ferait un très beau sujet de concours). ■

7.3 Dans la vie il y a 42 implications, ou comment se tordre la tête

"Soit n le plus petit entier naturel non définissable en moins de quinze mots".

Théorème 7.14. *Toute implication de la théorie $\vdash A \rightarrow B$ est démontrable.*

Si l'on veut démontrer une conclusion de la forme $\Gamma, A \vee B \vdash C$, on peut éliminer un \vee à partir des deux prémisses $\Gamma, A \vdash$ et $\Gamma, B \vdash C$, ce qui n'est pas une règle de la déduction naturelle. Pourtant, on pourrait étendre l'arbre pour fournir une preuve respectant les règles. On motive ainsi la définition suivante :

Définition 7.15 (Règle dérivable / démontrable). On dit qu'une règle de la forme

$$\frac{\Gamma_1 \vdash A_1 \quad \dots \quad \Gamma_n \vdash A_n}{\Delta \vdash B}$$

est *dérivable* ou *démontrable* s'il existe un fragment d'arbre de preuve dont la conclusion est $\Delta \vdash B$ et dont les prémisses sont les $\Gamma_i \vdash A_i$.

Remarque. Une règle est dérivable lorsqu'elle est démontrable avec les autres règles.

Exemple. Si l'on a $\Gamma \vdash A \rightarrow B$ alors

$$\frac{\Gamma \vdash A}{\Gamma \vdash B}$$

est dérivable, car il suffit de compléter par un arbre \mathcal{T} l'arbre de preuve

$$\frac{\frac{\mathcal{T}}{\Gamma \vdash A \rightarrow B} \quad \Gamma \vdash A}{\Gamma \vdash B} (\rightarrow e)$$

Définition 7.16 (Règle admissible). Une règle

$$\frac{\Gamma_1 \vdash A_1 \quad \dots \quad \Gamma_n \vdash A_n}{\Delta \vdash B} (\mathcal{R})$$

est *admissible* lorsqu'il existe une fonction φ sur les arbres de preuve qui à un arbre de preuve utilisant (\mathcal{R}) associe un arbre de même conclusion sans (\mathcal{R}) .

Exemple. La règle d'affaiblissement

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B} \text{ (affaiblissement)}$$

n'est pas démontrable avec les règles de déduction naturelle. Mais on se dit qu'en ajoutant A dans tous les contextes supérieurs, on obtient un arbre de preuve valide.

Remarque. Si une règle (\mathcal{R}) est dérivable, alors (\mathcal{R}) est admissible.

Lemme 7.17. *Si $\Gamma \vdash A$ alors $\Gamma, B \vdash A$.*

Démonstration. Par induction sur l'arbre de preuve :

- Si on a l'arbre $\frac{}{\Gamma \vdash A} \text{ (ax)}$ alors $A \in \Gamma$ donc $A \in \Gamma, B$ donc $\frac{}{\Gamma, B \vdash A} \text{ (ax)}$
- Si on a l'arbre

$$\frac{\Gamma, A \vdash A'}{\Gamma \vdash A \rightarrow A'} \text{ (}\rightarrow\text{i)}$$

alors par hypothèse d'induction $\Gamma, A \vdash A'$, donc il existe un arbre de preuve \mathcal{T} de $\Gamma, A, B \vdash A'$ si bien qu'on a l'arbre

$$\frac{\frac{\mathcal{T}}{\Gamma, B, A \vdash A'}}{\Gamma, B \vdash A \rightarrow A'} \text{ (}\rightarrow\text{i)}$$

- Etc...

■

Proposition 7.18. *La règle d'affaiblissement est admissible.*

Démonstration. On note $\Gamma \vdash A$ si on a un arbre de preuve en déduction naturelle et $\Gamma \vdash_a A$ lorsqu'on a un arbre de preuve en déduction AVEC affaiblissement. À partir d'un arbre de $\Gamma \vdash_a A$, on construit un arbre de preuve de $\Gamma \vdash A$. Par induction sur l'arbre de $\Gamma \vdash_a A$

- Si $\frac{}{\Gamma \vdash_a A} \text{ (ax)}$ alors $A \in \Gamma$ donc $\frac{}{\Gamma \vdash A} \text{ (ax)}$
- Etc...

■

7.4 Quantificateurs (1^{er} ordre)

On étend l'ensemble des formules grâce aux quantificateurs : pour une variable x et une formule φ , on définit $\forall x, \varphi$ et $\exists x, \varphi$ comme des formules. Ainsi, $\forall x, x$ est une formule. Il faut cependant faire attention à la capture, on a bien $\forall x, x = \forall y, y$ mais $\forall x, \forall y, x \rightarrow y \neq \forall y, \forall y, y \rightarrow y$, de même qu'en mathématique, on a

$$\int_0^x f(t) dt = \int_0^x f(u) du \quad \text{mais} \quad \int_0^{42} \int_0^1 f(x, y) dy dx \neq \int_0^{42} \int_0^1 f(y, y) dy dy$$

Commençons avec un exemple d' α conversion : $\forall x, x \rightarrow y =_\alpha \forall t, t \rightarrow y$

Définition 7.19 (Variable libre). Soit \mathcal{F} est une formule. on définit inductivement $FV(\mathcal{F})$ par :

- $FV(x) = \{x\}$
- $FV(\top) = FV(\perp) = \emptyset$
- $FV(\mathcal{F}_1 \vee \mathcal{F}_2) = FV(\mathcal{F}_1 \wedge \mathcal{F}_2) = FV(\mathcal{F}_1 \rightarrow \mathcal{F}_2) = FV(\mathcal{F}_1) \cup FV(\mathcal{F}_2)$
- $FV(\exists x, \varphi) = FV(\forall x, \varphi) = FV(\varphi) \setminus \{x\}$

Définition 7.20 (Substitution). Si φ et Ψ sont des formules, si x est une variable, on définit $\varphi[x \mapsto \Psi]$ par induction sur φ :

- $y[x \mapsto \Psi] = y$
- $x[x \mapsto \Psi] = \Psi$
- $(\mathcal{F}_1 \wedge \mathcal{F}_2)[x \mapsto \Psi] = \mathcal{F}_1[x \mapsto \Psi] \wedge \mathcal{F}_2[x \mapsto \Psi]$
- Idem pour $\vee, \rightarrow \dots$
- $(\forall y, \varphi)[x \mapsto \Psi] = \forall y, \varphi[x \mapsto \Psi]$ si $y \notin FV(\Psi)$

Exemple. $(\forall y, x \rightarrow y)[x \mapsto y] \neq \forall y, y \rightarrow y$

$$\forall y, x \rightarrow y =_{\alpha} \forall t, x \rightarrow t$$

$$\forall t, y \rightarrow t$$

$(\forall y, \varphi)[x \mapsto \Psi] = \forall t, \varphi[y \mapsto t][x \mapsto \Psi]$ où t est la nouvelle variable qui n'est ni dans φ ni dans Ψ : variable fraîche.

$$(\forall x, \varphi)[x \mapsto \Psi] = \forall x, \varphi$$

$$(\exists x, \varphi)[x \mapsto \Psi] = \exists x, \varphi \text{ (cette formule ne contient pas de } x \text{)}$$

Définition 7.21 (Règles pour le \forall).

Introduction du \forall :

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \forall x, \varphi} \text{ (Intro } \forall \text{) lorsque } x \notin FV(\Gamma)$$

Élimination du \forall :

$$\frac{\Gamma \vdash \forall x, \varphi}{\Gamma \vdash \varphi[x \mapsto t]} \text{ (Elim } \forall \text{)}$$

Définition 7.22 (Règles pour le \exists).

Introduction du \exists :

$$\frac{\Gamma \vdash \varphi[x \mapsto t]}{\Gamma \vdash \exists x, \varphi} \text{ (Intro } \exists \text{)}$$

Élimination du \exists :

$$\frac{\Gamma \vdash \exists x, \varphi \quad \Gamma, \varphi \vdash A}{\Gamma \vdash A} \text{ (Elim } \exists \text{) si } x \notin FV(\Gamma) \text{ et } x \notin FV(A)$$

Exemple. Une preuve non recevable : SCANDALE !

$$\frac{\overline{x \vdash x} \text{ (ax)}}{x \vdash \forall x, x} \text{ (Intro } \forall \text{)}$$

En effet, x est déjà une variable libre de Γ ! Pour sortir du \forall , il faudrait introduire une variable qui ne soit pas dans le contexte, mais la preuve devient rapidement bien plus dure :

$$\frac{\frac{?}{x \vdash y}}{x \vdash \forall x, x} \text{ (Intro } \forall \text{)}$$

Voici maintenant un exemple de preuve correcte :

$$\frac{\frac{\overline{x, x \vdash x} \text{ (ax)}}{\vdash x \rightarrow x} \text{ (Intro } \rightarrow \text{)}}{\vdash \forall x, x \rightarrow x} \text{ (Intro } \forall \text{)}$$

Exemple. Montrons $\neg(\forall x, \varphi) \rightarrow \exists x, \neg\varphi$ avec φ une formule.

$$\frac{\frac{\overline{\Gamma \vdash \neg(\forall x, \varphi)} \text{ (ax)}}{\Gamma := \neg(\forall x, \varphi), \neg(\exists x, \neg\varphi) \vdash \perp} \text{ (raa)}}{\vdash \neg(\forall x, \varphi) \rightarrow \exists x, \neg\varphi} \text{ (Intro } \rightarrow \text{)}$$

$$\frac{\frac{\frac{\overline{\Gamma, \neg\varphi \vdash \perp} \text{ (ax)}}{\Gamma \vdash \varphi} \text{ (Intro } \forall \text{)}}{\Gamma, \neg\varphi \vdash \perp} \text{ (raa)}}{\Gamma, \neg\varphi \vdash \neg(\exists x, \neg\varphi)} \text{ (Intro } \rightarrow \text{)}$$

$$\frac{\overline{\Gamma, \neg\varphi \vdash \neg(\exists x, \neg\varphi)} \text{ (ax)}}{\Gamma, \neg\varphi \vdash \exists x, \neg\varphi} \text{ (Elim } \rightarrow \text{)}$$

Exemple. Montrons que $\forall x, \varphi \wedge \psi \vdash (\forall x, \varphi) \wedge (\forall x, \psi)$:

$$\frac{\frac{\frac{\overline{\forall x, \varphi \wedge \psi \vdash \forall x, \varphi \wedge \psi} \text{ (ax)}}{\forall x, \varphi \wedge \psi \vdash \varphi \wedge \psi} \text{ (Elim } \wedge \text{)}}{\forall x, \varphi \wedge \psi \vdash \varphi} \text{ (Intro } \forall \text{)}}{\forall x, \varphi \wedge \psi \vdash \forall x, \varphi} \text{ (Intro } \wedge \text{)}$$

$$\frac{\frac{\frac{\overline{\forall x, \varphi \wedge \psi \vdash \forall x, \varphi \wedge \psi} \text{ (ax)}}{\forall x, \varphi \wedge \psi \vdash \varphi \wedge \psi} \text{ (Elim } \wedge \text{)}}{\forall x, \varphi \wedge \psi \vdash \psi} \text{ (Intro } \forall \text{)}}{\forall x, \varphi \wedge \psi \vdash \forall x, \psi} \text{ (Intro } \wedge \text{)}$$

$$\forall x, \varphi \wedge \psi \vdash (\forall x, \varphi) \wedge (\forall x, \psi)$$

Exercices

Exercice 7.1. En déduction naturelle, démontrer les séquents suivants :

- $\vdash a \vee b \rightarrow b \vee a$;
- $\vdash (a \rightarrow b) \rightarrow (a \vee c) \rightarrow (b \vee c)$;
- $\vdash a \vee (b \wedge c) \rightarrow (a \vee b) \wedge (a \vee c)$;
- $\vdash (a \wedge b) \wedge c \rightarrow a \wedge (b \wedge c)$;
- $\vdash (a \rightarrow b) \rightarrow (\neg b \rightarrow \neg a)$;
- $\vdash p \rightarrow \neg \neg p$.

Exercice 7.2 (Dédution naturelle classique). On considère les règles de la déduction naturelle intuitionniste, auxquelles on enlève la règle *efqs*, que l'on remplace par la règle *reductio ad absurdum* :

$$\frac{\Gamma, \neg a \vdash \perp}{\Gamma \vdash a} (raa)$$

Montrer en déduction naturelle classique les assertions suivantes :

$$\begin{array}{ccccccc} \vdash \neg \neg p \rightarrow p & \frac{\Gamma \vdash \perp}{\Gamma \vdash p} (\perp E) & \frac{\Gamma, \neg p \vdash p}{\Gamma \vdash p} & \vdash p \vee \neg p & \frac{\Gamma \vdash \neg q \rightarrow \neg p}{\Gamma \vdash p \rightarrow q} (contrap) \\ \vdash ((p \rightarrow q) \rightarrow p) \rightarrow p & \frac{\Gamma, \neg p \vee \neg q \vdash r}{\Gamma, \neg(p \wedge q) \vdash r} & \frac{\Gamma, \neg p, \neg q \vdash r}{\Gamma, \neg(p \vee q) \vdash r} & \frac{\Gamma, p, \neg q \vdash r}{\Gamma, \neg(p \rightarrow q) \vdash r} \end{array}$$

Exercice 7.3 (Dédution naturelle classique minimale). On note DNC la déduction naturelle classique. On note DNCM la partie de DNC n'utilisant que les connecteurs \neg et \rightarrow . Les jugements de DNC sont notés avec \vdash et ceux de DNCM avec \vdash_M .

On note V l'ensemble des variables. On définit une fonction f de l'ensemble des propositions de DNC dans l'ensemble des propositions de DNCM par :

$$\begin{aligned} f(p) &= p, p \in V \cup \{\perp\} & f(A \vee B) &= \neg f(A) \rightarrow f(B) & f(A \rightarrow B) &= f(A) \rightarrow f(B) \\ f(A \wedge B) &= \neg(f(A) \rightarrow \neg f(B)) \end{aligned}$$

L'objectif est de démontrer que $\vdash \varphi$ si et seulement si $\vdash_M f(\varphi)$.

- Que peut-on dire de $\Gamma \vdash \varphi$ si $\Gamma \vdash_M \varphi$ est dérivable dans DNCM ?
- Dans DNCM, montrer la règle d'affaiblissement et la règle réciproque de l'introduction de la flèche.
- Dériver les théorèmes suivants :

$$\vdash_M \neg(a \rightarrow b) \rightarrow a \qquad \vdash_M \neg(a \rightarrow \neg b) \rightarrow b$$

- Montrer les règles suivantes :

$$\frac{\Gamma, a \vdash_M b}{\Gamma, \neg b \vdash_M \neg a} \qquad \frac{\Gamma \vdash_M \neg a \rightarrow b \quad \Gamma, a \vdash_M c \quad \Gamma, b \vdash_M c}{\Gamma \vdash_M c}$$

- Montrer que si $\Gamma \vdash \varphi$ alors $f(\Gamma) \vdash_M f(\varphi)$ par induction sur l'arbre de dérivation de $\Gamma \vdash \varphi$.
- Montrer que si $\vdash f(\varphi) \rightarrow \varphi$ alors on a : $\vdash_M f(\varphi)$ implique $\vdash \varphi$.
- Montrer que $\vdash f(\varphi) \rightarrow \varphi$ par induction sur φ .
- Conclure.
- Interpréter ce résultat. Quelles en sont les conséquences en terme de langage, d'arbres de preuve, de technique de preuve, etc. ?

Exercice 7.4 (Modèles de Kripke). On considère l'ensemble des formules propositionnelles \mathcal{F} sur les variables \mathcal{V} et sur les connecteurs logiques \wedge, \vee et \rightarrow .

On notera $\Gamma \vdash_i \varphi$ et $\Gamma \vdash_c \varphi$ pour indiquer qu'un séquent $\Gamma \vdash \varphi$ est prouvable en logique intuitionniste et classique respectivement.

— Pour chaque des formules suivantes, donner un arbre de preuve en logique classique :

- $\neg\neg p \rightarrow p$,
- $((\neg p) \rightarrow p) \rightarrow p$.

— Montrer que $(\neg\varphi \vee \psi) \rightarrow (\varphi \rightarrow \psi)$ est prouvable en logique intuitionniste.

On appelle modèle de Kripke un triplet $\mathcal{K} = (S, \leq, \Vdash)$ tel que :

- U est un ensemble, appelé *univers*, dont les éléments sont appelés des *mondes*,
- \leq est préordre sur U , c'est-à-dire une relation réflexive et transitive,
- \Vdash est une relation binaire, appelée *forçage*, entre les éléments de S et de \mathcal{V} telle que :

$$\forall a, b \in S^2, \forall x \in \mathcal{V}, \text{ si } a \leq b \text{ et } a \Vdash x \text{ alors } b \Vdash x$$

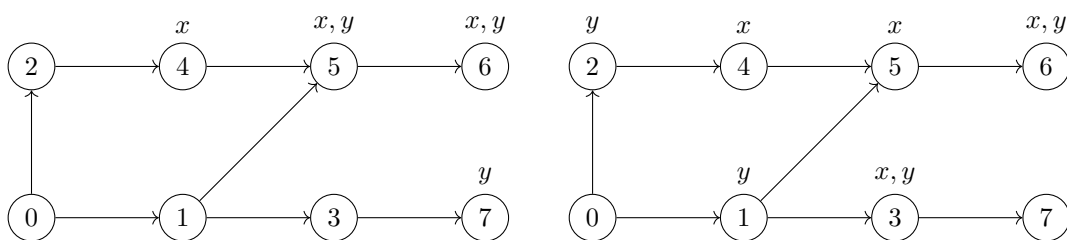
On étend \Vdash à une relation binaire entre les éléments de S et les formules par :

- pour $a \in S, a \Vdash \top$ et $a \nVdash \perp$;
- pour $a \in S$ et $\varphi, \psi \in \mathcal{F}^2$:
 - $a \Vdash \varphi \wedge \psi$ si et seulement si $a \Vdash \varphi$ et $a \Vdash \psi$;
 - $a \Vdash \varphi \vee \psi$ si et seulement si $a \Vdash \varphi$ ou $a \Vdash \psi$;
 - $a \Vdash \varphi \rightarrow \psi$ si et seulement si pour tout $b \in S$, si $a \leq b$ et $b \Vdash \varphi$, alors $b \Vdash \psi$.

Pour $\varphi \in \mathcal{F}$ et $\Gamma \subseteq \mathcal{F}$, on note :

- $a \Vdash \Gamma$ si pour tout $\psi \in \Gamma, a \Vdash \psi$;
- $\mathcal{K} \Vdash \varphi$ (resp. $\mathcal{K} \Vdash \Gamma$) si pour tout $a \in S, a \Vdash \varphi$ (resp. $a \Vdash \Gamma$) ;
- $\Gamma \Vdash \varphi$ si pour tout modèle de Kripke $\mathcal{K} = (S, A, \Vdash)$ et $a \in S$, si $a \Vdash \Gamma$, alors $a \Vdash \varphi$;
- $\Vdash \varphi$ si $\emptyset \Vdash \varphi$ ou si pour tout $\mathcal{K}, \mathcal{K} \Vdash \varphi$. On dit alors que φ est vraie.

Pour $\mathcal{V} = \{x, y\}$, déterminer, parmi les deux représentations suivantes, laquelle correspond à un modèle de Kripke. Justifier.



Pour le modèle de Kripke correspondant, indiquer quels sommets réalisent la formule $y \vee \neg x$.

- Montrer que pour $x \in \mathcal{V}, x \vee \neg x$ n'est pas une formule vraie.
- Montrer que les formules de la première question ne sont pas des formules vraies.
- Soit $\varphi \in \mathcal{F}, \mathcal{K} = (S, A, \Vdash)$ un modèle de Kripke et $a, b \in S^2$. Montrer que si $a \Vdash \varphi$ et $a \leq b$, alors $b \Vdash \varphi$.
- En déduire que la logique intuitionniste est correcte pour la sémantique de Kripke, c'est-à-dire que si $\Gamma \vdash_i \varphi$, alors $\Gamma \Vdash \varphi$. Que peut-on en déduire sur la logique intuitionniste dans la sémantique booléenne usuelle ?

Exercice 7.5 (Le retour de Kripke). Un ensemble de formules Δ est *saturé* si $\Delta \nVdash \perp$ et si, pour toutes formules φ et Ψ , si $\Delta \vdash \varphi \vee \Psi$ alors $\varphi \in \Delta$ et $\Psi \in \Delta$.

- Soit φ une formule et Γ un ensemble de formules tel que $\Gamma \nVdash \varphi$. Montrer que Γ est contenu dans un ensemble saturé tel que $\Delta \nVdash \varphi$.

- Soit Δ_0 un ensemble saturé. On définit le triplet $M = \langle U, \subseteq, I \rangle$ où U est l'ensemble des ensembles saturés contenant Δ_0 et $I(a) = \{\Delta \supseteq \Delta_0 \mid a \in \Delta\}$. Montrer que ceci définit un modèle de Kripke et que dans ce modèle on a, pour tout ensemble saturé Δ contenant Δ_0 et pour toute formule $\varphi : \Delta \Vdash \varphi$ si et seulement si $\varphi \in \Delta$.
- En déduire la complétude de la logique intuitionniste.

Exercice 7.6 (Théorie des groupes). On considère la théorie des groupes, c'est-à-dire les formules comprenant des variables et les opérateurs e , \times et $^{-1}$, munies des règles de déduction suivantes :

$$\begin{array}{cccc}
\frac{}{x = x} (Eg) & \frac{x = y}{y = x} (Sym) & \frac{x = y \quad y = z}{x = z} (Trans) & \frac{x = y \quad x' = y'}{x \times x' = y \times y'} (Eq \times) \\
\\
\frac{x = y}{x^{-1} = y^{-1}} (EqI) & \frac{}{(x \times y) \times z = x \times (y \times z)} (Assoc) & \frac{}{x \times e = x} (E1) & \frac{}{e \times x = x} (E2) \\
\\
\frac{}{x \times x^{-1} = e} (I1) & & \frac{}{x^{-1} \times x = e} (I2) &
\end{array}$$

Montrer les énoncés suivants :

$$\begin{array}{cccc}
e = e^{-1} & (x \times y) \times (y^{-1} \times x^{-1}) = e & \frac{x \times y = e}{y = x^{-1}} & (x \times y)^{-1} = y^{-1} x^{-1} \\
\\
\frac{(a \times b) \times (a \times b) = (a \times a) \times (b \times b)}{a \times b = b \times a}
\end{array}$$

Si on suppose que pour tout z on a $z \times z = e$, alors montrer $x \times y = y \times x$. Ceci est-il formulable comme une règle de déduction en premier ordre ?

Exercice 7.7. Méditer rêveusement sur les questions suivantes en regardant vaguement les murs :

1. Vous saviez que dans n'importe quel bar non vide, il existe une personne telle que si elle boit, toutes les autres personnes dans le bar boivent ?
2. On peut montrer qu'il existe deux nombres irrationnels a et b tels que a^b soit rationnel. En effet, on montre facilement que soit $a = b = \sqrt{2}$ soit $(a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2})$ convient. Est-ce gênant de ne pas savoir duquel des deux couples il s'agit ?
3. Quel est le plus petit entier non définissable en moins de douze mots ?
4. On ne retient de la langue française que les adjectifs, dont on précise bien le sens pour qu'il n'y ait pas d'ambiguïté. On dira qu'un adjectif est autodoxe s'il se décrit lui-même (par exemple *ín court ž* ou *ín polysyllabique ž*), antidoxe sinon (par exemple *ín long ž* ou *ín monosyllabique ž*). *ín* Antidoxe *ž* est-il autodoxe ou bien antidoxe ?
5. La semaine prochaine, il y aura un examen surprise. Tellement surprise, en fait, que vous ne pourrez jamais savoir quel jour il aura lieu, avant le dernier moment. Est-ce que ça pose problème ?

Chapitre 8

Grammaires

Sommaire

8.0	Banalités	107
8.1	Définitions	108
8.2	Lemme fondamental des grammaires non contextuelles	109
8.3	Ambiguïté dans les grammaires	111
8.4	Arbre d'analyse	112
8.5	Les trucs que je n'ai pas le droit de vous raconter parce que c'est écrit hors programme, mais je le fais quand même	112
8.5.1	Décider si $u \in L(\mathcal{G})$	112
8.5.2	Automates à pile	113
8.5.3	Lemme de l'étoile pour les langages algébriques (Exercice 8.5)	113
	Exercices	113

La grammaire, ça m'intéresse
qu'en informatique

Paul Raphael

8.0 Banalités

Historiquement, les grammaires formelles sont issues de recherche linguistique des années soixante-dix, notamment par Noam CHOMSKY, et se sont par la suite développées sous l'influence de la naissante informatique théorique avec des chercheurs comme Marcel-Paul SCHÜTZENBERGER. Lorsque l'on utilise T_EX, qui a été conçu avant la maturité des grammaires, l'utilisation d'une commande produit une substitution. Si on définit `\def\truc{blah \x blah}`, alors l'appel à `\truc` dans le corps du texte sera remplacé à la compilation par `blah \x blah`. Ainsi, T_EX, avant de compiler le document, modifie le code source.

Le C est un langage qui arrive après les grammaires, donc il ne fait donc pas que des remplacements, mais il raisonne « par blocs », ou plus prédestinent des arbres de syntaxe.

```
int main() {  
    if(_)  
    {  
        -  
    }  
    int x = 42;  
    return zi;  
}
```

Un compilateur, comme ceux du C, lit une chaîne de caractère et décide de la validité de celle-ci (c'est-à-dire être dans le langage) et construit alors son arbre de syntaxe abstraite. C'est, d'un certain point de vue, un automate. On voit ce phénomène également en mathématiques.

En revanche le processeur de C marche toujours par substitution, donc les *#include* et les *#define* (macros) modifie le code avant de le donner au compilateur. Pour cette raison, l'utilisation des macros est hors programme et en général déconseillé.

Exemple. L'alphabet $\Sigma := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, (,), *\}$ permet de construire le langage des expressions arithmétiques. Ainsi, 42 , $1 + 1$, $2 * 2 + 1$ sont des éléments de celui-ci. En revanche, $** + 3$ n'en est pas un. [fix pour toi ;)]

Pour reconstruire une expression à partir de son arbre de syntaxe revient à faire un parcours préfixe de celui-ci. En notation infixe, on perd le sens des priorités et donc nous sommes forcés de re-parenthésier les expressions. Un écueil disparu avec la notation postfixe (qui s'appelle notation en *polonaise inversée*, utilisé par les calculatrices Casio d'antan). Il s'agit donc de comprendre le comportement infixe des arbres de syntaxe, c'est le but des grammaires.

8.1 Définitions

Définition 8.1 (Grammaire, HP). Une *grammaire* est un quadruplet (Σ, N, S, P) où :

- Σ est un alphabet appelé *ensemble des symboles terminaux*
- N un alphabet tel que $N \cap \Sigma = \emptyset$, appelé *ensemble des symboles non terminaux*
- $S \in N$ appelé *symbole initial*
- $P \subset ((\Sigma \sqcup N)^* \setminus \Sigma^*) \times ((\Sigma \sqcup N)^*)$, la *règle de production*.

On notera souvent P par une flèche \rightarrow .

Remarque. Par convention, les lettres de Σ seront souvent notées en minuscule, et celles de N souvent en majuscule.

Exemple. Prenons les règles de production $S \rightarrow aBb$; $B \rightarrow Saa$; $aB \rightarrow S$ qui encodent la grammaire $(\{a, b\}, \{S, B\}, S, \{-, -, -\})$

On peut alors définir une relation de *dérivation* sur $(\Sigma \sqcup N)^*$:

Définition 8.2 (Dérivation immédiate). Soit $u \in (\Sigma \sqcup N)^*$. S'il existe $(x, y, z) \in ((\Sigma \sqcup N)^*)^3$ avec $y \rightarrow w \in P$, on écrit

$$u = xyz \Rightarrow xwz$$

On aimerait pouvoir dire que, si $u \Rightarrow v \Rightarrow w$ alors $u \Rightarrow w$, et qu'un mot se dérive lui-même, ce qui motive la définition suivante :

Définition 8.3 (Dérivation). La dérivation est la clôture transitive, réflexive de la dérivation immédiate \Rightarrow et est notée \Rightarrow^* (c'est la plus petite relation transitive qui contient \Rightarrow).

C'est à dire que $u \Rightarrow^* v$ lorsqu'il existe $n \in \mathbb{N}$ et des mots u_0, \dots, u_n tels que $\forall i, u_i \Rightarrow u_{i+1}$ et $u_0 = u$ et $u_n = v$

Remarque. Lorsque la dérivation s'effectue en $n \in \mathbb{N}$ étapes, on note également $u \Rightarrow^n v$.

On se donne, à partir de maintenant, une grammaire $\mathcal{G} := (\Sigma, N, S, P)$.

Définition 8.4 (Mot accepté par une grammaire). Un mot $u \in \Sigma^*$ est accepté par la grammaire \mathcal{G} si $S \Rightarrow^* u$.

Définition 8.5 (Langage engendré par une grammaire). On note $L(\mathcal{G})$ langage des mots acceptés par la grammaire \mathcal{G} que l'on qualifie d'*engendré* par celle-ci. Formellement :

$$L(\mathcal{G}) := \{u \in \Sigma^* : S \Rightarrow^* u\}$$

Exemple (Mots bien parenthésés). Si on prend la grammaire définie par

$$S \rightarrow \varepsilon \quad (1)$$

$$S \rightarrow SS \quad (2)$$

$$S \rightarrow aSb \quad (3)$$

On a donc :

$$ab \in L(\mathcal{G}) \text{ car } S \xrightarrow{(3)} aSb \xrightarrow{(1)} ab.$$

Également, $a^3b^3 \in L(\mathcal{G})$, ou encore $a^n b^n \in L(\mathcal{G})$ car : $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \dots$

On note L_p le langage des mots bien parenthésés dont on rappelle la définition inductive (voir l'exercice 1.2).

$$L_p = \{\varepsilon\} \cup L_p \cdot L_p \cup aL_p b.$$

Montrons par double inclusion que $L_p = L(\mathcal{G})$

\supseteq Soit $u \in L(\mathcal{G})$. Il existe donc $n \in \mathbb{N}$ tel que $S \Rightarrow^n u$. Montrons par récurrence forte sur $n \in \mathbb{N}$ que $u \in L_p$.

- Initialisation : si $S \Rightarrow^0 u$ avec $u \in \Sigma^*$. Puisque $S \notin \Sigma$, la seule dérivation possible est donc la (1). Par conséquent $u = \varepsilon \in L_p$.
- Hérédité : soit $n \in \mathbb{N}^*$ tel que $\forall u \in \Sigma^*, (\exists k \in \llbracket 0, n-1 \rrbracket, S \Rightarrow^k u) \implies (u \in L_p)$. Soit donc $u \in \Sigma^*$ tel que $S \Rightarrow^n u$. Si $u = \varepsilon$ alors par définition $u \in L_p$. Puisque $n \geq 1$, distinguons ensuite :
 - Ou bien, $S \Rightarrow SS \Rightarrow^{n-1} u$ (auquel cas d'ailleurs, $n \geq 2$). Or le lemme fondamental garantit l'existence de u_1, u_2 deux mots et de n_1, n_2 entiers naturels vérifiant $n_1 + n_2 = n - 1$ tel que $S \Rightarrow^{n_1} u_1, S \Rightarrow^{n_2} u_2$ avec $u = u_1 u_2$. Remarquons alors que $n_1 \leq n - 1$ et $n_2 \leq n - 1$ de sorte que l'hypothèse de récurrence appliquée à u_1 et u_2 montrent que ces derniers sont dans L_p si bien que par définition inductive de ce même ensemble, $u \in L_p$.
 - Ou bien, $S \Rightarrow aSb \Rightarrow^{n-1} u$ qui se traite de même que le cas ci-dessus.

Ainsi, dans tout les cas $u \in L_p$ et le résultat au rang n est ainsi démontré.

\subseteq Soit $u \in L_p$ On construit une dérivation $S \Rightarrow^* u$ par induction sur u

- Si $u = \varepsilon$, alors $S \Rightarrow^* \varepsilon$ immédiatement.
- Si $u = avb$. D'après l'hypothèse d'induction, $S \Rightarrow^* v$ car $v \in L_p$ donc $S \Rightarrow aSb \Rightarrow^* avb$.
- Si $u = vw$ où $vw \in L_p$, alors $S \Rightarrow SS \xRightarrow{HI} vS \Rightarrow^* vw \in L(\mathcal{G})$.

8.2 Lemme fondamental des grammaires non contextuelles

Définition 8.6 (Grammaire non contextuelle). Une *grammaire non contextuelle* est un quadruplet (Σ, N, S, P) où :

- Σ est un alphabet appelé *ensemble des symboles terminaux*
- N un alphabet tel que $N \cap \Sigma = \emptyset$, appelé *ensemble des symboles non terminaux*
- $S \in N$ appelé *symbole initial*
- $P \subset N \times ((\Sigma \sqcup N)^*)$, la *règle de production*.

On notera souvent P par une flèche \rightarrow .

Les éléments de P sont les productions de la grammaire. Σ s'appelle l'ensemble des terminaux. N s'appelle l'ensemble des non terminaux.

Remarque. De l'anglais *context free*, on trouve également la dénomination *hors contexte*.

Jusqu'à la fin du chapitre, toutes les grammaires considérées seront non contextuelles. Par ailleurs, si on vous propose de vous baigner dans un volcan amazonien, ou de travailler avec des grammaires contextuelles, préférez un bon maillot de bain.

Théorème 8.7 (Lemme fondamental). Soit $u := u_1 \cdots u_p \in (\Sigma \sqcup N)^*$ où $u_i \in (\Sigma \sqcup N)^*$. Si $u \Rightarrow^n v$ alors il existe $(v_1, \dots, v_p) \in ((\Sigma \sqcup N)^*)^p$, $(m_1, \dots, m_p) \in \mathbb{N}^n$ tels que :

- $\forall i \in \llbracket 1, n \rrbracket, u_i \Rightarrow^{m_i} v_i$
- $\sum_{i=1}^n m_i = n$
- $v = v_1 \dots v_p$

Démonstration. Raisonnons par récurrence sur n :

- Pour $n = 0$, le résultat est évident.
- Pour $n \geq 1$, on a : $u_1 \cdots u_p \Rightarrow^1 w \Rightarrow^{n-1} v$. $u_1 \cdots u_p \Rightarrow^1 w$ s'obtient en appliquant une des règles de la grammaire $X \rightarrow m$. Donc il existe i tel que $u_i = u'_i X u''_i$. Donc la première dérivation c'est $u_1 \cdots u_{i-1} u'_i X u''_i u_{i+1} \cdots u_p \Rightarrow u_1 \cdots u_{i-1} u'_i m u''_i u_{i+1} \cdots u_p$. D'après l'hypothèse de récurrence appliquée à $w \Rightarrow^{n-1} v$, il existe v_1, \dots, v_p et n_1, \dots, n_p tels que :
 - $\forall k \neq i, u_k \Rightarrow^{n_k} v_k$
 - $u'_i m u''_i \Rightarrow^{n_i} v_i$
 - $v_1 \cdots v_p = v$
 - $\sum_{k=1}^p n_k = n - 1$

Alors si $k \neq i, u_k \Rightarrow^0 u_k$ donc $u_k \Rightarrow^{n_k} v_k$ et $u'_i X u''_i \Rightarrow^1 u'_i m u''_i \Rightarrow^{n_i} v_i$ donc :

- $u_1 \cdots u_p \Rightarrow^n v_1 \cdots v_p = v$
- $\forall k \neq i, u_k \Rightarrow^{n_k} v_k$ et $u_i \Rightarrow^{n_i+1} v_i$ et on a bien $\sum_{k \in \llbracket 1, p \rrbracket \setminus \{i\}} n_k + (n_i + 1) = n$

■

Définition 8.8 (Langage algébriques). On dit que un langage est *algébrique* s'il existe une grammaire non contextuelle qui engendre ce langage. L'ensemble des langages algébriques sur un alphabet Σ est noté $\text{Alg}(\Sigma)$.

Proposition 8.9. Les langages algébriques sont stables par union, concaténation et étoile.

Démonstration. Soient L_1, L_2 deux langages algébriques de grammaires associées $G_1 = (\Sigma_1, N_1, S_1, P_1)$ et $G_2 = (\Sigma_2, N_2, S_2, P_2)$.

- Pour l'union, posons $G := (\Sigma_1 \sqcup \Sigma_2, N_1 \sqcup N_2 \sqcup \{S\}, S, P_1 \sqcup P_2 \sqcup \{S \rightarrow S_1 | S_2\})$. On montre par double inclusion que $L(G) = L_1 \cup L_2$, les deux sens sont faciles.
- Pour la concaténation, posons $G := (\Sigma_1 \sqcup \Sigma_2, N_1 \sqcup N_2 \sqcup \{S\}, S, P_1 \sqcup P_2 \sqcup \{S \rightarrow S_1 S_2\})$.
 - \supseteq Soit $u \in L_1 \cdot L_2$. Il existe $(v, w) \in L_1 \times L_2$ tels que $u = vw$. $v \in L_1$ donc dans $G_1 : S_1 \Rightarrow^* v$ et $w \in L_2$ donc dans $G_2 : S_2 \Rightarrow^* w$...
 - \subseteq Soit $u \in L(G)$ alors $S \Rightarrow^* u$. Regardons la première étape. D'après la définition de G c'est $S : \Rightarrow S_1 S_2 \Rightarrow^* u$. D'après le lemme fondamental des grammaires non contextuelles il existe v, w dans Σ^* tels que $S_1 \Rightarrow^* v$ et $S_2 \Rightarrow^* w$ et $u = vw$. Donc $S_1 \Rightarrow^* v$ dit que $v \in L(G_1)$ et $S_2 \Rightarrow^* w$ dit que $w \in L(G_2)$. Donc $u \in L_1 \cdot L_2$.
- Pour l'étoile, posons $G := (\Sigma_1 \sqcup \Sigma_2, N_1 \sqcup N_2 \sqcup \{S\}, S, P_1 \sqcup P_2 \sqcup \{S \rightarrow S_1 S | \varepsilon\})$. L'idée est : par récurrence forte, si $S \Rightarrow^k u$ alors $u \in L^*$.

■

Corollaire 8.10. On a l'inclusion $\text{Reg}(\Sigma) \subset \text{Alg}(\Sigma)$.

Démonstration. En effet, $\text{Reg}(\Sigma)$ est le plus petit ensemble de langages stable par union, concaténation et étoile. On peut fournir une seconde preuve à l'aide d'automates : Soit $\mathcal{A} := (\Sigma, Q, q_I, F, \delta)$. Construisons une grammaire (Σ, N, S, P) qui engendre $L(\mathcal{A})$. On pose

- $N := \{X_q\}_{q \in Q}$
- $S := X_{q_I}$
- Dans $\mathcal{A} : q \xrightarrow{a} q'$ donne dans G une production : $X_q \rightarrow a X_{q'}$ et pour $X_q \in F$ on a la règle $X_q \rightarrow \varepsilon$

Montrons que $L(G) = L(\mathcal{A})$:

- \supseteq Si $u \in L(\mathcal{A})$, il existe un chemin dans $\mathcal{A} : q_0 \xrightarrow{u_0} q_1 \rightarrow \cdots \xrightarrow{u_{n-1}} q_n \in F$. Dans G :

$$S = X_{q_I} \Rightarrow u_0 X_{q_1} \Rightarrow \cdots \Rightarrow u_0 \cdots u_{n-1}$$

donc $u \in L(G)$.

- \subseteq Réciproquement, on montre par récurrence que si $S \Rightarrow^* w \in (\Sigma \cup N)^*$ alors w est de la forme $w_0 \cdots w_{n-1} w_n$. Par récurrence, on montre que si $S \Rightarrow w_0 \cdots w_{n-1} X_q$ alors $\delta^*(q_I, w_0 \cdots w_{n-1}) = q$ ce qui prouve que $u \in L(\mathcal{A})$.

■

8.3 Ambiguïté dans les grammaires

Exemple (A-rithmétique). $S \rightarrow a|S + S|S \times S$. L'expression $a + a \times a$ est engendrée par cette grammaire. On a plusieurs possibilités :

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S \times S \Rightarrow^2 a + a \times a \quad (V1)$$

$$S \Rightarrow S \times S \Rightarrow S + S \times S \Rightarrow a + S \times S \Rightarrow a + a \times S \Rightarrow a + a \times a \quad (V2)$$

$$S \Rightarrow S \times S \Rightarrow S + S \times S \Rightarrow S + a \times S \Rightarrow a + a \times S \Rightarrow a + a \times a \quad (V2')$$

Les deux dernières dérivations sont les mêmes à des échanges bénins de règles. On peut passer de V2 à V2' par échange de règles qui, d'après le lemme fondamental, sont à des places indépendantes. MAIS pas de V1 à V2.

Définition 8.11 (Dérivation la plus à gauche). $X \Rightarrow_g Y$ lorsqu'il existe $(X_1, X_2, X_3) \in ((\Sigma \sqcup N)^*)^3$ tel que $X = X_1 X_2 X_3$ où $X_1 \in \Sigma^*$, $X_2 \in N$, $X_3 \in (\Sigma \cup N)^*$ et une production $X_2 \rightarrow Y_2$ et $Y = X_1 Y_2 X_3$.

Exemple (Retour sur l'A-rithmétique). $S \Rightarrow_g S + S \Rightarrow_g a + S \Rightarrow_g a + a$ est une dérivation la plus à gauche correcte. Voici maintenant une dérivation qui ne suit pas cette règle : $S \Rightarrow_g S + S \not\Rightarrow_g S + a$

V2 est la dérivation la plus à gauche obtenue à partir de V2' en échangeant quelques étapes qui d'après le lemme fondamental sont indépendantes. En revanche ce n'est pas la même chose pour V1 et V2 qui sont des dérivations les plus à gauches différentes.

Définition 8.12. Une grammaire est *ambiguë* lorsqu'il existe un mot $u \in \Sigma^*$ qui se dérive, le plus à gauche, de deux façons différentes.

Proposition 8.13. $S \Rightarrow^* u$ si et seulement si $S \Rightarrow_g^* u$: la dérivation à gauche engendre les mêmes mots.

Démonstration. Par double implication :

\Leftarrow Évident.

\Rightarrow Soit $u_1 \in \Sigma^*$, alors $u_1 X u_2 \Rightarrow_g u_1 Y u_2$. Par hypothèse récurrence, $u_1 Y u_2 \Rightarrow^{n-1} v$. Soit $Z \in N$ tel que $u_1 = u'_1 Z u''_1$ où $u'_1 \in \Sigma^*$.

D'après le lemme fondamental, $u'_1 Z u''_1 X u_2 \Rightarrow^n V_1 V_2 V_3 V_4 V_5 = V$. On a $u'_1 \in \Sigma^*$ donc $v_1 = u'_1$

$$— Z \Rightarrow^{n_2} V_2$$

$$— u''_1 \Rightarrow^{n_3} V_3$$

$$— X \Rightarrow^{n_4} V_4$$

$$— u_2 \Rightarrow^{n_5} V_5$$

$$\text{donc } u_2 \Rightarrow_g^{n_2} u'_1 V_2 u''_1 X u_2 \Rightarrow_g^{n_3+n_4+n_5} u_1 V_2 V_3 V_4 V_5 \text{ (car } u'_1 \in \Sigma^*)$$

■

Exemple. On peut donc "corriger" l'A-rithmétique :

$$— S \rightarrow S + S$$

$$— S \rightarrow M$$

$$— M \rightarrow M \times M$$

$$— M \rightarrow a$$

Cette nouvelle grammaire n'est pas ambiguë.

Exemple. — $S \rightarrow S + S$

$$— S \rightarrow M$$

$$— M \rightarrow M \times M | a | (S)$$

Théorème 8.14. Soit G une grammaire algébrique. " G est-elle ambiguë ?" est un problème de décision indécidable.

Démonstration. Correspondance de Post (simplifiée) : $(x_1, \dots, x_m) \in (\Sigma^*)^m$ et $(y_1, \dots, y_m) \in (\Sigma^*)^m$. Existe-t-il des indices tels que ...

■

8.4 Arbre d'analyse

Exemple. Soit $S \rightarrow a|S + S|S * S$. Le mot $a + a * a * a$ est dérivable de la façon suivante :

$$S \Rightarrow S * S \Rightarrow S * S * S \Rightarrow S + S * S * S \Rightarrow^4 a + a * a * a \quad (1)$$

Mais on peut le dériver d'autres manières, comme

$$S \Rightarrow S * S \Rightarrow S * S * S \Rightarrow S * S * a \Rightarrow S + S * S * a \Rightarrow^3 a + a * a * a \quad (2)$$

ou encore

$$S \Rightarrow S + S \Rightarrow S + S * S \Rightarrow S + S * S * S \Rightarrow^4 a + a * a * a \quad (3)$$

Ainsi, chaque dérivation peut se représenter par un arbre :

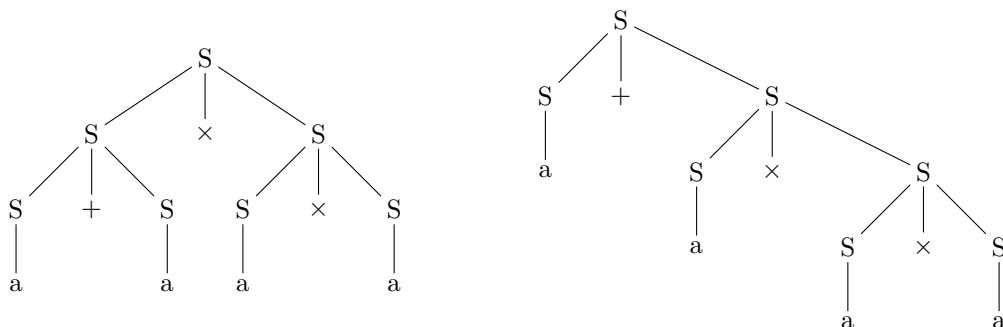


FIGURE 8.1 – À gauche, l'arbre des dérivations (1) et (2), à droite celui de (3)

Nous avons vu ici une définition avec les mains, formalisons un peu ce nouvel objet.

Définition 8.15. Si $X \xRightarrow{n} Y$, avec $X \in N$ et $Y \in (N \cup \Sigma)^*$ on construit l'arbre d'analyse de $X \xRightarrow{n} Y$ par récurrence sur n :

- Pour $n = 0$, $X \xRightarrow{0} X$ on prend l'arbre qui a seulement le noeud X .
- Sinon, $X \xRightarrow{1} \alpha_1 \cdots \alpha_k \xRightarrow{n-1} Y$. D'après le lemme fondamental, il existe y_1, \dots, y_k et n_1, \dots, n_k tel que $\forall i, \alpha_i \xRightarrow{n_i} y_i$ et $\sum_{i=1}^k n_i = n - 1$. On a des arbres d'analyse T_1, \dots, T_k des $\alpha_i \xRightarrow{n_i} y_i$. On construit l'arbre avec X comme racine, d'enfants T_1, \dots, T_k .

Proposition 8.16. Une grammaire est ambiguë si et seulement s'il existe deux arbres d'analyses distincts pour un même mot.

La preuve est technique, pénible à rédiger, nous ne la ferons pas. On notera en revanche qu'obtenir la dérivation la plus à gauche d'un mot à partir de son arbre d'analyse avec un parcours en profondeur en prenant les fils de chaque noeud de gauche à droite.

Exemple. $S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * S * S \Rightarrow a + a * a * S \Rightarrow a + a * a * a$

Définition 8.17 (Mot de feuilles). Si T est un arbre d'analyse, son mot de feuilles, noté $\mathcal{L}(T)$, est le mot défini par induction sur l'arbre :

- si x est un noeud sans enfant : il n'a rien compris au réarmement démographique, $\mathcal{L}(x) = x$
- si T est un noeud d'enfants T_1, \dots, T_n (dans cet ordre), alors $\mathcal{L}(T) = \mathcal{L}(T_1) \cdot \mathcal{L}(T_2) \cdots \mathcal{L}(T_n)$.

8.5 Les trucs que je n'ai pas le droit de vous raconter parce que c'est écrit hors programme, mais je le fais quand même

8.5.1 Décider si $u \in L(\mathcal{G})$

On s'intéresse, dans cette section, au problème de reconnaissance d'un mot par une grammaire. On utilisera les formes normales de CHOMSKY (exercice 4) et GREIBACH (exercice 10).

Exemple (Exercice 4). 1. $X \rightarrow \varepsilon$

$X \Rightarrow^* \varepsilon$.

$Y \rightarrow \dots X \dots$. On ajoute $Y \rightarrow \dots \neq \varepsilon$

2. $A \rightarrow B$. Pour chaque A , soit $E_A = \{B \in N : A \Rightarrow^* B\}$. Pour chaque règle qui contient A à droite : $X \rightarrow \alpha_1 A \alpha_2 A \alpha_3$. On ajoute $X \alpha_1 (X_1 \in E_A) \alpha_2 (X_2 \in E_A) \alpha_3 \dots$. Puis, on retire toutes les productions $A \rightarrow B$.

Il faut donc «Chomskyser» la grammaire (c'est-à-dire la mettre sous forme normale de CHOMSKY) :
 $A \rightarrow BC$

$A \Rightarrow a$

$S \rightarrow \varepsilon$.

Si G est propre, comment peut on la Chomskyser ? Soit $X \Rightarrow \alpha_1 \dots \alpha_k$ une production de cette grammaire autre que $S \rightarrow \varepsilon$. Si c'est $X \rightarrow a$, on ne fait rien. Si c'est $X \rightarrow N_1 N_2$, on ne fait rien. Ce n'est pas $X \rightarrow N$ car la grammaire est propre. Donc, $k \geq 2$. Pour chaque lettre $a \in \Sigma$, on crée un non terminal $L_a \rightarrow a$ avec cette règle, et on remplace chaque lettre a dans les $\alpha_1 \dots \alpha_k$. On a maintenant une règle $X \rightarrow \tilde{\alpha}_1 \dots \tilde{\alpha}_k$, où $\tilde{\alpha}_i \in N$. On la remplace par les règles $X \rightarrow \tilde{\alpha}_1 C_1$ Puis $C_1 \rightarrow \tilde{\alpha}_2 C_2 \dots C_{k-2} \rightarrow \tilde{\alpha}_{k-1} \tilde{\alpha}_k$

8.5.2 Automates à pile

Ce sont les sextuplets $(\Sigma, N, Q, q_I, F, \delta)$ avec $\delta : N \times \Sigma \times Q \rightarrow \{N, \text{push}, \text{pop}\} \times Q$.

Ceci définit un automate. Un mot est accepté par cet automate si et seulement s'il arrive sur un état final avec une pile vide. On peut montrer que les automates à pile reconnaissent exactement les langages algébriques. Et avec deux piles ? Ca, ça s'appelle une machine de Turing.

8.5.3 Lemme de l'étoile pour les langages algébriques (Exercice 8.5)

Si $X \rightarrow \alpha_1 \dots \alpha_k$, on dit que le terme de droite est de longueur k

Le nombre de noeuds au ième étage d'un arbre de hauteur au plus i avec à chaque noeud au plus m enfants est m^i . Donc $|w| \leq m^i$. Dans l'arbre de dérivation, de ?, il existe une branche de longueur au moins $k + 1$. Il existe une branche qui contient au moins $k + 1$ non terminaux, donc il existe un non terminal A qui apparaît 2 fois. Soit n_1 le noeud X du haut et n_2 le noeud X du bas. Soit x le préfixe du mot de feuille de l'arbre formé des feuilles à gauche de n_1 . On pose $z =$ le mot de feuille à droite de n_1 , et $y =$ le mot de feuille de n_2 , $v =$ le mot de feuille de n_1 composé uniquement des feuilles à gauche de celles de n_2 , et $w =$ à droite. On a bien $u = xvywz$. Si $|vw| = 0$, $X \Rightarrow^* \alpha X \beta$, où $\alpha \Rightarrow^* v$ et $\beta \Rightarrow w$. Si $|vw| = 0$, alors $\alpha, \beta \Rightarrow^* \varepsilon$ ce qui est exclu car la grammaire est propre (les non terminaux à droite n'engendrent pas ε et seul S peut engendrer ε). Donc $|vw| \geq 1$

On choisit dans la branche la plus longue le non terminal X répété tel que la première occurrence est à profondeur maximale. Donc la hauteur de l'arbre n_1 est plus petite que k (car si la branche est S_k , il existe un non terminal qui se répète dans cette branche ce qui contredit le fait que n_1 est de profondeur maximale) donc $|vyw| \leq m^k$ d'après la question 1. D'après l'arbre, $S \Rightarrow^* xXy \Rightarrow^* xvXwz$ car $X \Rightarrow^* vXw \uparrow$ (arbre n_1) et (arbre n_2) $X \Rightarrow^* y$. \uparrow par récurrence dit : $\forall i \in \mathbb{N}^*, X \Rightarrow^* v^i X w^i$.

$X \Rightarrow^* v^i X w^i$. Or, $X \Rightarrow^* vXw$ donc $X \Rightarrow^* v^i vXw w^i$ or $X \Rightarrow^* X = v^0 X w^0$. Or, $X \Rightarrow^* y$ donc $\forall i \in \mathbb{N}, X \Rightarrow^* X = v^i X w^i$. D'où le résultat.

Exercices

Exercice 8.1. Donner des grammaires non contextuelles engendrant les langages suivants :

- $\{a^i b^j c^k \mid i > j\}$;
- $\{a^i b^j c^k \mid i \neq j\}$;
- $\{a^i b^j c^k \mid i \neq j \text{ ou } j \neq k\}$;
- l'ensemble des mots sur $\{a, b\}$ ayant le même nombre de a que de b ;
- l'ensemble des mots sur $\{a, b\}$ ayant deux fois plus de a que de b ;
- $\{w \# \bar{w} \# \mid w \in \{a, b\}^*\}$;
- l'ensemble des mots sur $\{a, b\}$ qui ne sont pas de la forme ww .

Exercice 8.2. Quel est le langage engendré par la grammaire suivante ?

$$S \rightarrow aS \mid aSbS \mid \varepsilon$$

Exercice 8.3 (Stabilité des langages algébriques). Montrer que l'ensemble des langages algébriques est stable par union, par concaténation et par étoile.

Exercice 8.4 (Formes normales). Une grammaire G est dite *propre* lorsque les trois conditions suivantes sont satisfaites :

1. (ε -libre) si $\varepsilon \in L(G)$ alors ε n'apparaît que dans une production $S \rightarrow \varepsilon$;
2. il n'y a pas de production contenant S à droite ;
3. il n'y a pas de production (dite *simple*) de la forme $A \rightarrow B$.

Montrer que l'on peut transformer toute grammaire en une grammaire propre.

Une grammaire en forme normale de CHOMSKY est une grammaire propre dont toutes les productions sont de la forme :

$$A \rightarrow BC \quad \text{ou} \quad A \rightarrow a \quad \text{ou} \quad S \rightarrow \varepsilon$$

Soit G une grammaire engendrant un langage L . Montrer qu'il existe une grammaire en forme normale de CHOMSKY qui engendre le langage $L \setminus \{\varepsilon\}$.

Exercice 8.5 (Lemme de l'étoile algébrique). On veut prouver le résultat suivant :

Lemme 8.18. *Si L est un langage algébrique, il existe un entier N tel que pour tout mot $u \in L$, si $|u| \geq N$ alors il existe des mots x, v, y, w, z tels que :*

- $u = xvywz$;
- $|vw| \geq 1$;
- $|vyw| \leq N$;
- $\forall i \geq 0, xv^i y w^i z \in L$.

Soit G une grammaire propre. On note m la longueur maximale d'un membre droit de la grammaire.

1. On considère un arbre de dérivation, on suppose que toutes les branches sont de longueur inférieure ou égale à i . Soit w le mot de feuille de l'arbre. Montrer que $|w| \leq m^i$.
2. Soit k le nombre de non terminaux de la grammaire G . Soit $N = m^k + 1$ et soit u un mot de $L(G)$ tel que $|u| \geq N$. Montrer qu'un arbre de syntaxe dont le mot de feuille est u contient un chemin sur lequel deux noeuds sont étiquetés par un même non littéral A .
3. Conclure.

Exercice 8.6 (Application du lemme de l'étoile). 1. Montrer que le langage $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ n'est pas algébrique.

2. Montrer que l'ensemble des langages algébriques n'est pas stable par intersection.
3. L'est-il par complémentaire ?

Exercice 8.7 (Lemme d'Ogden). L'objectif est de montrer une version plus forte du lemme de l'étoile pour les langages algébriques.

Lemme 8.19. *Soit L un langage algébrique. Il existe un entier N tel que pour tout mot $z \in L$ dans lequel on marque au moins N positions distinctes, il est possible de décomposer z sous la forme $z = uxvyw$ avec :*

- x ou y contient au moins une position marquée,
- xvy contient au plus N positions marquées,
- pour tout $i \geq 0, ux^i v y^i w \in L$.

On utilise une grammaire en forme normale de CHOMSKY (CNF).

Définition 8.20. Soit T un sous-arbre d'un arbre de dérivation selon une grammaire CNF. On suppose marquées certaines feuilles de T . On appelle *embranchement* un noeud de T ayant deux fils, tel que chacun de ses fils contienne au moins une feuille marquée.

1. Soit T un sous-arbre d'un arbre de dérivation selon une grammaire CNF. On suppose qu'au moins 2^h feuilles distinctes de T ont été marquées. Montrer qu'il existe un chemin, d'une feuille à la racine, passant par au moins h embranchements et tel que pour tout i , le i -ème embranchement ait au plus 2^i descendants marqués.
2. On considère une grammaire CNF G engendrant le langage L . Montrer qu'il existe un entier N tel que :
 - pour tout mot $w \in L$ dans lequel on marque au moins N positions,
 - pour tout arbre de dérivation de w , il existe deux embranchements b_1 et b_2 tels que
 - b_1 est un ancêtre de b_2 ,
 - b_1 est un ancêtre d'au plus N feuilles marquées,
 - b_1 et b_2 sont étiquetés par la même variable.
3. En déduire le lemme d'Ogden.

Exercice 8.8 (Application du lemme d'Ogden). On s'intéresse au langage $L = \{a^i b^j c^k d^l \mid i = 0 \text{ ou } j = k = l\}$.

1. Montrer que pour tout $N \in \mathbb{N}$ et tout mot $z \in L$ il existe une décomposition $z = uxvyyw$ telle que :
 - $|xy| \geq 1$,
 - $|xvy| \leq N$
 - pour tout $i \geq 0$, $ux^i v y^i w \in L$.
2. Montrer que L n'est pas algébrique.

Exercice 8.9. Montrer que si L est algébrique et si R est rationnel, le langage $L \cap R$ est algébrique.

Exercice 8.10 (Forme normale de GREIBACH). Une grammaire est en forme normale de GREIBACH lorsque toutes ses productions sont de la forme $A \rightarrow a\alpha$ où $a \in \Sigma$ et $\alpha \in N^*$ (un mot de non terminaux).

1. On considère des productions :

$$A \rightarrow \alpha_1 B \alpha_2 \quad \text{et} \quad B \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_p$$

Montrer que l'on peut obtenir une grammaire équivalente n'utilisant pas le symbole B marqué à droite de la première production ci-dessus.

2. On considère des productions :

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_n \quad \text{et} \quad A \rightarrow \beta_1 \mid \cdots \mid \beta_p$$

Construire une grammaire équivalente dans laquelle les non terminaux n'apparaissent qu'en dernière position de chaque production (sans tenir compte des non terminaux qui pourraient être présents dans les α_i et les β_j).

3. Montrer que toute grammaire est faiblement équivalente à une grammaire en forme normale de GREIBACH.

Chapitre 9

Apprentissage et I.A.

Sommaire

9.0	Dans la vie, tout est un vecteur de \mathbb{R}^n	117
9.1	Apprentissage supervisé	117
9.1.1	Algorithme des k plus proches voisins (k -NN)	117
9.1.2	Matrice de confusion	120
9.1.3	ID3 et arbres de décision	120
9.2	Apprentissage non supervisé	121
9.2.1	CHA : Classification hiérarchique et ascendante	121
9.2.2	Algorithme des k -moyennes	122
9.3	Algorithme A*	122

Le chapitre ou on sait pas ce qu'on
va faire ni comment on va le faire

F.Hatat

9.0 Dans la vie, tout est un vecteur de \mathbb{R}^n

On vit dans \mathbb{R}^n muni de sa norme euclidienne standard. Imaginons des points de \mathbb{R}^n formant des patates imaginaires. L'être humain y voit 3 patates tandis que l'ordinateur y voit seulement des points. Si un point se trouve dans un endroit que l'humain n'associe à aucune patate, il peut être difficile pour l'ordinateur de lui donner un quelconque appartenance à une patate, faut-il alors lui en créer une ?

Le problème peut se poser dans la vie de tous les jours avec la reconnaissance de calligraphie : quelle lettre désigne un caractère tracé à la main ?

On souhaite avoir une fonction qui va de \mathbb{R}^n dans un ensemble fini de classes. Par exemple, pour la poste – devant envoyer une lettre et devant lire l'adresse – les caractères peuvent être transformés en un tableau de pixels et voilà nos premiers vecteurs. Il faut ensuite la fonction renvoyant un vecteur sur la classe de sa lettre.

9.1 Apprentissage supervisé

9.1.1 Algorithme des k plus proches voisins (k -NN)

Définition 9.1 (Algorithme avec apprentissage). On a un jeu de données $(x_k, c_k)_k$ où $x_k \in \mathbb{R}^n$ et $c_k \in C$ un ensemble fini de classes. On a une fonction (qu'on ne connaît pas) $\Phi : \mathbb{R}^n \rightarrow C$. On souhaite trouver un algorithme permettant d'approcher la fonction Φ telle qu'il nous fournisse $f : \mathbb{R}^n \rightarrow C$ tel que " $f(x) = \Phi(x)$ assez souvent".

Définition 9.2 (k -NN). L'algorithme des k plus proches voisins fait partie de la famille des algorithmes avec apprentissage. C'est un algorithme de classification d'un objet par rapport à un ensemble de classes dans un espace normé. Sa complexité est en la taille des données d'entraînement multiplié par le coût de la mise à jour des k meilleurs voisins actuels.

Algorithme 9.1 Algorithme des k plus proches voisins

Entrée : $x \in \mathbb{R}^n$

Sortie :

Pour chaque j , **faire** :

On calcule $\|x - x_j\|$

Fin pour

On garde uniquement les k valeurs (x_1, \dots, x_k) qui sont les plus proches de x . Parmi les classes des x_1, \dots, x_k on attribue à x celle qui est majoritaire.

La complexité de l'algorithme des k plus proches voisins est en

$O(\text{taille des données d'entraînement} \times \text{coût de la mise à jour des } k \text{ meilleurs voisins actuels})$

Si on a N points pour s'entraîner, cela donnerait une complexité en $O(N \times k)$. On pourrait rajouter la complexité du calcul de la norme, mais cela dépend de n , qui est une constante du problème et non un paramètre.

Remarque. En utilisant un tas binaire, on peut faire descendre la complexité des kNN à $O(N \cdot \log(k))$. (cf cours de première année).

On rappelle ici que l'insertion est en temps $O(\log(k))$ où k est le nombre d'éléments de l'arbre (on percole). Pour la suppression, on est en même complexité. (en rappelant qu'on ne peut supprimer que la racine, par définition du tas binaire). Il faut ici avoir un tas maximum, le pire des kNN se trouve au sommet. On peut alors le supprimer et le substituer par le pire de ses fils, puis on percole vers le bas le nouvel arrivant. Cependant, l'intérêt reste théorique, en pratique, il n'est pas très intéressant si k est suffisamment petit.

Exemple. Illustrons maintenant l'algorithme sur un exemple. On veut prédire la classe de l'objet ■ :

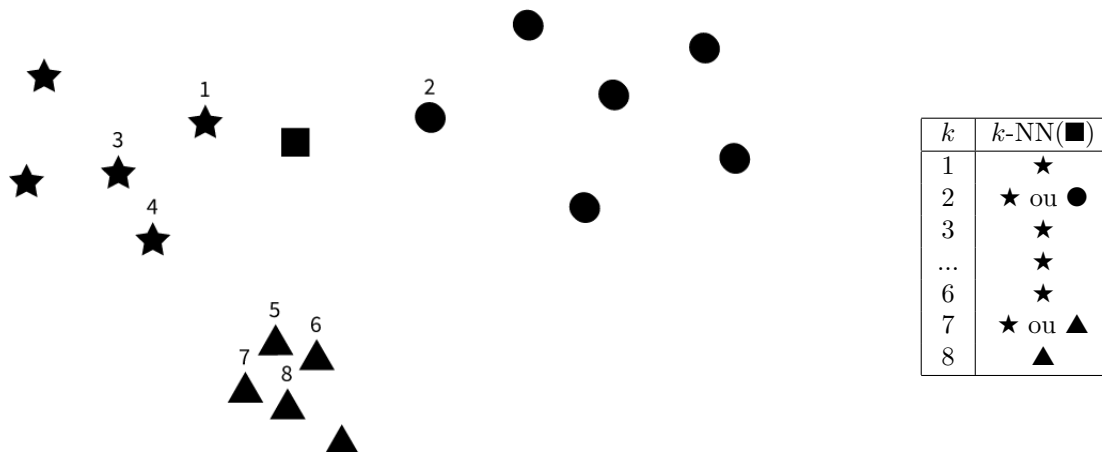


FIGURE 9.1 – Exemple des k plus proches voisins

Remarque. L'exemple ci dessus illustre bien que la classe renvoyée par l'algorithme dépend énormément du nombre k . Il n'existe pas de "bonne réponse" absolue.

Exemple (MNIST). MNIST (cf. TP 11) est une base de donnée encodant des images (28×28 p) composée d'images d'entraînement, il y en a 30000. Le but est de reconnaître des caractères à partir d'images.

Définition 9.3 (Arbres k dimensionnels). On partitionne l'espace des données d'entraînement. On utilise les médianes des espaces partitionnés pour repartitionner récursivement les sous ensemble de notre espace. Cela se traduit par un arbre appelé arbre k -dimensionnel.

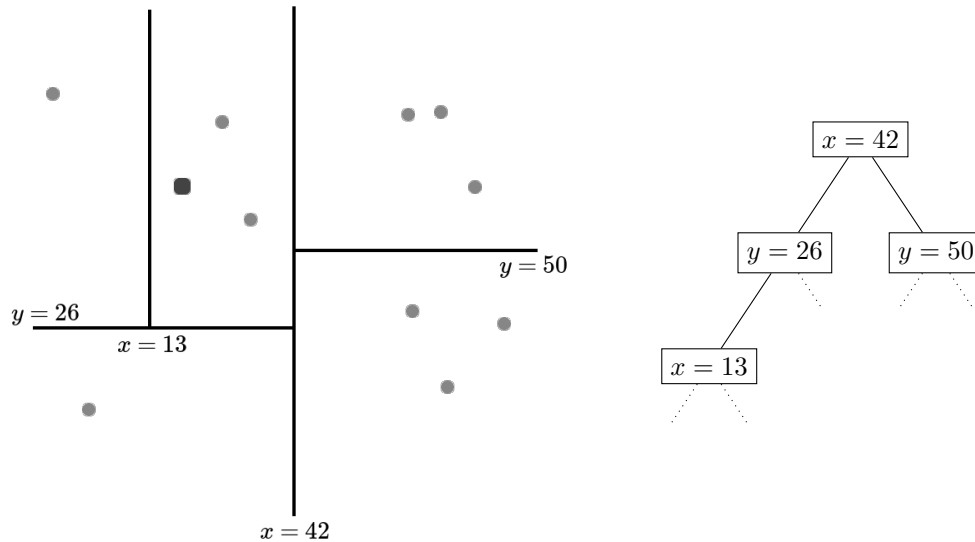


FIGURE 9.2 – Exemple d'un arbre k dimensionnel

Construction de l'arbre : On partitionne les N points des données d'entraînement (dans \mathbb{R}^n) par médiane de leur k -ème coordonnée x_k . On crée un noeud

On construit récursivement fg en partitionnant les données de gauche par médiane sur la $(k + 1 \bmod n)$ -ème coordonnée (avec n le nombre de dimensions). On construit fd de même et on recommence.

Cela donne, grâce au choix de la médiane, un arbre de hauteur $O(\log N)$

Mais, ça foire : dans l'arbre k dimensionnel, rien ne garantit que le plus proche voisin du vecteur à classifier (ou même un plus grand nombre des plus proches voisins) sera dans la même branche de l'arbre. On l'imagine bien en prenant un objet classifié très proche de la bordure fixée par la médiane.

Algorithme 9.2 Recherche des k -NN dans un arbre dimensionnel.

Entrée : un vecteur $x = (x_1, \dots, x_n) \in \mathbb{R}^n$, un arbre dont la racine est une médiane sur la dimension d

Sortie : k plus proches voisins de x parmi ceux de tout l'arbre

fonction KNN(k , racine) :

Si $x_d < \text{racine}$ **alors**

$p_1, \dots, p_k \leftarrow \text{KNN}(k, \text{racine.fils_xgauche})$ les k plus proches voisins de x dans le fils gauche.

Test ▷ Il est possible d'en trouver moins si il n'y en a pas assez

$\delta \leftarrow \max_{1 \leq i \leq k} \|x - p_i\|$ ▷ La distance entre x et le voisin trouvé le plus loin

Fin si

Si $x_d \geq \text{racine}$ **ou** $B(x, \delta) \not\subset \{(y_1, \dots, y_n) \in \mathbb{R}^n : y_d \leq \text{racine}\}$ **ou** on a trouvé moins de k voisins

alors

Chercher q_1, \dots, q_k les k plus proches voisins de x dans le fils droit.

Fin si

Si On a moins de k voisins trouvés récursivement **alors**

On va chercher ceux qui manquent dans l'autre fils

Fin si

renvoyer Les k plus proches voisins parmi $p_1, \dots, p_k, q_1, \dots, q_k$

Fin fonction

Remarque. Ici, dans le pire cas on explore tout l'arbre en $O(N)$, on a donc théoriquement rien gagné. Mais si on a de la chance, on est en $O(\log N)$.

9.1.2 Matrice de confusion

On a un algorithme qui classe des vecteurs de \mathbb{R}^n dans des classes C_1, \dots, C_p . On lance l'algorithme sur des vecteurs (dont on connaît par ailleurs la bonne classe) et on construit la matrice d'ordre $p \times p$ tel que $M_{i,j}$ = le nombre de vecteurs classés C_i alors que la vraie classe était C_j .

Si notre algorithme est parfait, on aura une matrice diagonale. A l'inverse, le pire algorithme du monde aura une matrice avec que des 0 en diagonale.

9.1.3 ID3 et arbres de décision

Exemple (Akinator). Une application très classique de ce qui va suivre est le jeu Akinator : un "génie" pose une série de questions pour deviner le personnage auquel vous pensez. Ceci est en fait un arbre de décision. L'algorithme fournit également une stratégie pour le Qui-est ce

On a N données stockées dans S , avec $S = \bigsqcup_{k=1}^n C_k$. Ici, on ne travaille plus dans \mathbb{R}^n , mais plutôt dans un espace de type $K_1 \times \dots \times K_n$ où chacun des K_i est un ensemble fini. Informellement, chacune des dimensions de \mathbb{R}^n devient finie (et donc en particulier discrète).

Définition 9.4 (Entropie de Shannon). On définit l'entropie comme :

$$H(S) = - \sum_{i=1}^p \frac{\#C_i}{N} \log \frac{\#C_i}{N}$$

Proposition 9.5 (Entropie minimale). Si toutes les classes sont vides sauf une,

$$H(S) = 0$$

ce cas est appelé cas d'entropie minimale.

Proposition 9.6 (Entropie maximale). L'entropie maximale est atteinte lorsque $\frac{\#C_i}{N} = \frac{1}{p}$. Lorsque les classes sont toutes de la même taille, c'est ce qui maximise le désordre. On a alors :

$$H(S) = - \log\left(\frac{1}{p}\right) = \log(p)$$

Toujours dans \mathbb{R}^n , pour $x \in \mathbb{R}$, on note $S_{x_k=x} := \{v \in S | v_k = x\}$. Dans le jeu d'entraînement, on n'a que p_1, \dots, p_m valeurs possibles par x_k . On pose

$$G(S, k) = H(S) - \sum_{i=1}^m p_i H(S_{x_k=p_i})$$

qu'on définit comme le gain d'information de la k -ème composante.

Exemple (On joue au golf?). En fonction de la météo, de la température, de l'humidité, du vent et on se demande si on a envie d'aller jouer au golf.

- $K_1 = \{\text{ensoleillé, nuageux, pluvieux}\} = \{-1, 0, 1\}$
- $K_2 = \{\text{chaud, moyen, froid}\} = \{-1, 0, 1\}$
- $K_3 = \{\text{élevée, normale}\} = \{0, 1\}$
- $K_4 = \{\text{fort, faible}\} = \{0, 1\}$

L'ensemble de départ est donc $K_1 \times K_2 \times K_3 \times K_4$, et on arrive dans $\{0, 1\}$.

Ainsi, le gain apporté par l'attribut "Météo" est :

$$G(S, \text{météo}) = H(S) - \sum_{x \in K_1} \frac{\#(S_{\text{météo}=x})}{\#S} H(S_{\text{météo}=x})$$

Informellement, si on sent que fixer un certain critère permet de ranger tout le monde dans la même classe, ce critère va être très rangeant et va donc avoir un gain d'information très élevé. Donc, si on suppose que pour tous les golfeurs (ou la majorité), connaître la météo leur permet déjà de savoir s'ils vont aller jouer au golf (c'est à dire que le gain de la météo est le plus grand), on mettrait donc la météo en racine de notre arbre.

Algorithme 9.3 Algorithme ID3

fonction ID3(S) :

Si $S = \emptyset$ **alors**

 On fabrique une feuille qui choisit comme classe la classe majoritaire dans le parent

Sinon, si S n'a que des éléments d'une même classe C_i

 On fabrique une feuille C_i

Sinon

 On cherche la dimension k non utilisée parmi les ancêtres entre 1 et n qui maximise $G(S, k)$

 On construit un noeud de valeur k ayant pour fils p_1, \dots, p_q .

 On partitionne S en $P_1 \sqcup \dots \sqcup P_q$

Fin si

Pour i allant de 1 à q , **faire** :

$p_i = \text{ID3}(P_i)$

▷ Construction récursive sur les fils

Fin pour

Fin fonction

9.2 Apprentissage non supervisé

On se donne N vecteurs. En combien de classes les partitionner, et comment ?

9.2.1 CHA : Classification hiérarchique et ascendante

Algorithme 9.4 Classification hiérarchique et ascendante

Entrée : Des vecteurs de \mathbb{R}^n

Sortie : Des classes partitionnant les vecteurs

On commence par mettre chaque vecteur dans sa classe.

Tant que mon critère préféré totalement arbitraire est vrai, **faire** :

 On fusionne les deux classes les plus proches

Fin tant que

On peut avoir comme critère préféré des choses comme :

- On a plus de k classes (on s'arrête donc quand on a k classes)
- La distance minimale entre classes est encore assez petite (on s'arrête quand continuer regrouperait des classes éloignées).

Distance entre deux classes ?

- Pour les mathématiciens, $d(C_1, C_2) = \min_{x_1 \in C_1, x_2 \in C_2} d(x_1, x_2)$.
- Mais certains informaticiens diraient $d(C_1, C_2) = \max_{x_1, x_2} d(x_1, x_2)$.
- Ou encore $d(C_1, C_2) = d(\text{barycentre de } C_1, \text{barycentre de } C_2)$.
- $d(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{(x_1, x_2) \in C_1 \times C_2} d(x_1, x_2)$.
- Distance de Ward :

$$\sqrt{\frac{|C_1||C_2|}{|C_1| + |C_2|}} \sum_{(x_1, x_2) \in C_1 \times C_2} d(x_1, x_2)$$

On notera qu'une propriété raisonnable attendue sur les distances est ce qu'on appelle la monotonie :

$$\min(d(C_i, C_j), d(C_i, C_k)) \leq d(C_i, C_j \cup C_k)$$

On remarque que ça rate pour les barycentres.

9.2.2 Algorithme des k -moyennes

Algorithme 9.5 k -moyennes

Entrée : N points à partitionner, k le nombre de classes voulues

Sortie : Les k classes remplies des points

On tire au hasard k points x_1, \dots, x_k . Ils représentent les classes C_1, \dots, C_k .

On répartit les autres points dans les classes C_1, \dots, C_k en mettant chaque x dans la classe C_j tel que $\|x - x_j\|$ est minimale.

Tant que les classes changent, **faire** :

C_1, \dots, C_k partitionnent désormais les N points. Le représentant de C_j est désormais le barycentre des points C_j (qui n'est pas nécessairement dedans !!).

On reclasse les N points du départ dans la classe C_j tel que $\|x - b_j\|$ est minimale.

Fin tant que

9.3 Algorithme A*

Pour étudier A*, il faut être à l'aise avec Dijkstra. 2.4.2

Notation : si $u \in V$, $d(u)$ représente la distance connue de src vers u .

On se donne une fonction h (une heuristique) tel que $h : V \rightarrow \mathbb{R}^+$ tel que $h(u)$ est une estimation de la distance qu'il reste à parcourir de u à $dest$. On note $w(u, v)$ le poids de l'arête de u à v

Algorithme 9.6 Algorithme A*

Entrée : Un graphe G , un point de départ src et un point d'arrivée $dest$, une heuristique h

Sortie : La distance entre src et $dest$

$P \leftarrow$ File de priorité

ENFILER($P, (src, 0)$)

Tant que P n'est pas vide, **faire** :

$u \leftarrow$ DÉFILER(P)

Pour chaque voisin v SI on a amélioré $d(u)$, **faire** :

$d(v) \leftarrow \min(d(v), d(u) + w(u, v))$

Si on a amélioré $d(v)$ **alors**

ENFILER($P, (v, \min(prio(v), d(v) + h(v))$)

Fin si

Fin pour

Fin tant que

Remarque. On peut donc redéfinir Dijkstra à partir de A* : Dijkstra = A* en prenant $h = 0$

Finalement, l'algorithme A* repose énormément sur son heuristique. Pour "classifier" les bonnes heuristiques, on pose la définition suivante :

Définition 9.7 (Heuristique admissible). Une heuristique h est *admissible* lorsque pour tout sommet v , $h(v)$ est un minorant de la distance de v à $dest$

Exemple. Dans un graphe plongé dans le plan, la distance euclidienne est une heuristique admissible.

Théorème 9.8. Avec une heuristique admissible, A* calcule $d(dest)$ correctement, dès que $dest$ est visité pour la première fois.

Démonstration. Soit δ la distance trouvée lorsque l'algorithme visite $dest$ pour la première fois. Par l'absurde, on suppose qu'il existe un autre chemin $src = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_n = dest$ plus court, de longueur δ' optimale.

Quand on visite s , on enfile v_1 avec la priorité $w(v_0, v_1) + h(v_1)$. h étant admissible, $h(v_1)$ sous-estime la distance optimale de v_1 à $dest$. Or, le chemin de $src = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_n = dest$ étant de longueur optimale, donc le chemin de $v_1 \rightarrow \dots \rightarrow v_n = dest$ est aussi optimal. Donc $w(src, v_1) + h(v_1) \leq \delta'$.

Plus tard, A* empile v_2 , avec priorité $p_2 \leq d(v_1) + w(v_1, v_2) + h(v_2) \leq \delta < \delta'$ (l'enfiler depuis v_1 est une des manières de l'enfiler, on pourrait trouver mieux par la suite).

Tous les sommets v_0, \dots, v_n sont insérés dans la file avec priorité $p_n \leq \delta'$. Donc en particulier, $dest = v_n$ est enfilée avec une priorité $p_n \leq \delta'$. Or A^* a visité la première fois $dest$ au moment où $d(dest) = \delta$. Donc $dest$ avait été enfilé avec une priorité $\delta + h(dest)$. Or h est admissible donc $0 \leq h(dest) \leq dist(dest, dest) = 0$. Donc $\delta' \leq \delta = p_n \leq \delta'$ Donc A^* a bien trouvé le chemin optimal au premier passage sur le sommet $dest$. D'où le résultat. ■

Mais, même avec une heuristique admissible, A^* a encore toutes les chances d'être exponentiel. Heureusement, il existe un moyen de garantir une complexité non exponentielle :

Définition 9.9 (Heuristique monotone). Une heuristique est *monotone* lorsque

$$\forall (u, v) \in V^2, h(u) \leq d(u, v) + h(v)$$

Proposition 9.10. Si h est monotone et $h(dest) = 0$, alors h est admissible.

Démonstration. Considérons le chemin $p : src = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_n = dest$.

$$w(p) = \sum_{i=0}^{n-1} d(v_i, v_{i+1}) \geq \sum_{i=0}^{n-1} h(v_i) - h(v_{i+1}) = h(v_0) - h(dest) = h(v_0)$$

En particulier pour le chemin optimal, $h(v_0) \leq \delta$. ■

Proposition 9.11. Si h est monotone, alors A^* possède une complexité en $O(\#E \log(\#V))$

Démonstration. On considère le chemin optimal $src \rightarrow \dots \rightarrow u \xrightarrow{w(u,v)} v$ Quand v sort pour la première fois de la file, v avait été enfilé avec une priorité majorée par $d(v) + h(v)$. $d(v) = d(u) + w(u, v)$. Et,

$$d(v) + h(v) = d(u) + w(u, v) + h(v) \tag{9.12}$$

$$\geq d(u) + h(u) \tag{9.13}$$

Sur le chemin de src à v , les sommets ont été enfilés par priorités croissantes le long du chemin. A^* fait alors sauter les sommets de la file dans l'ordre dans lequel ils apparaissent sur le chemin optimal. Par récurrence sur le nombre de sommets, quand u sort de la file, $d(u)$ est connu est bien la distance cherchée. Aux itérations suivantes, on n'améliore aucun $d(u)$ parmi ceux sortis de la file. La file contient au plus n éléments, les opérations sont en $O(\log n)$. On retrouve la complexité en $O(\#E \log(\#V)) = O(n^2 \log n)$ ■

Démonstration. ■

Table des algorithmes

0.1	Calcul de l'attracteur	14
0.2	Élagage Alpha-Bêta	16
2.1	Reconnaissance d'un arbre	29
2.2	Trouver un chemin améliorant	31
2.3	Trouver le couplage de plus grand cardinal	31
2.4	Recherche (bis) d'un tri topologique	34
2.5	Algorithme de KOSARAJU	36
2.6	Algorithme de calcul d'un arbre couvrant (<i>Union-Find</i>)	39
2.7	Algorithme de KRUSKAL	40
2.8	Algorithme de DIJKSTRA v.1	42
2.9	Algorithme de DIJKSTRA v.2	42
2.10	Algorithme de FLOYD-WARSHALL v.1	43
2.11	Algorithme de FLOYD-WARSHALL v.2	44
2.12	Algorithme de FLOYD-WARSHALL v.3 (c'est lui le "vrai")	45
3.1	Algorithme de PETERSON	49
3.2	Algorithme de PETERSON généralisé	50
3.3	Boulangerie de LAMPORT	50
3.4	Diner des philosophes	54
3.5	Producteurs consommateurs	54
3.6	Barrière de synchronisation	55
3.7	Protocole du coiffeur	59
3.8	Protocole de chaque client	59
4.1	Réduction de 3-SAT à Chemin hamiltonien	69
4.2	Algorithme qui recherche une correspondance de POST	71
6.1	Simuler un tirage sans remise de n entiers dans un ensemble $\llbracket 0, n - 1 \rrbracket$	93
6.2	RandQuickSort	96
6.3	Mélange de KNUTH FISHER-YATES	96
9.1	Algorithme des k plus proches voisins	118
9.2	Recherche des k -NN dans un arbre dimensionnel	119
9.3	Algorithme ID3	121
9.4	Classification hiérarchique et ascendante	121
9.5	k -moyennes	122
9.6	Algorithme A*	122
A.1	Glouton sur un matroïde	190
E.1	Max-Clique : retour sur trace	202
E.2	Max-clique : Bron Kerbosch	202

Références

- [1] Thibaut BALABONSKI et al. *Informatique : Cours et exercices corrigés : MP2I MPI*. Ellipses, 2022. ISBN : 9782340070349.
- [2] Thomas H. CORMEN et al. *Introduction to Algorithms*. 3^e éd. The MIT Press, 2009. ISBN : 0262033844.
- [3] Michael R. GAREY et David S. JOHNSON. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. USA : W. H. Freeman & Co., 1979. ISBN : 0716710447.
- [4] Maurice HERLIHY et Nir SHAVIT. *The Art of Multiprocessor Programming*. 1^{re} éd. Réédition. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2012. ISBN : 9780123973375.
- [5] John HOPCROFT, Jeffrey ULLMAN et Rajeev MOTWANI. *An Introduction to Automata Theory, Languages, and Computation*. 3^e éd. Pearson, 2006.
- [6] Christophe RAFFALLI, René DAVID et Karim NOUR. *Introduction à la Logique, Théorie de la démonstration*. 2004.
- [7] Robert SEDGEWICK et Kevin WAYNE. *Algorithms*. 4^e éd. Addison-Wesley Professional, 2015. ISBN : 0134384687.

Deuxième partie

Travaux pratiques

TP 0

Tas binomiaux

On étudie dans ce sujet une structure de données, les *tas binomiaux*, qui permet de réaliser une file de priorité. Les opérations que l'on souhaite réaliser sur cette structure sont :

- l'ajout d'un élément au tas binomial,
- la fusion de deux tas binomiaux,
- la suppression de l'élément minimal du tas,
- la modification de la valeur d'un élément présent déjà dans le tas.

0.1 Préparation de l'arborescence de travail

Question 0.1. Créez un répertoire `binomialHeap` à l'emplacement favori de votre dossier personnel. Dans ce répertoire, nous travaillerons ensuite essentiellement avec les trois fichiers suivants :

- `binomialHeap.ml` contiendra toutes les fonctions de manipulation des tas binomiaux ;
- `test.ml` contiendra un jeu de tests de ces fonctions ;
- `main.ml` servira de point d'entrée principal pour les applications de la fin de ce sujet.

On rappelle que le compilateur OCaml s'invoque de la façon suivante en ligne de commande :

```
$ ocamlc -c binomialHeap.ml
```

Ceci produit un fichier `binomialHeap.cmo`. Ce n'est pas un fichier exécutable, c'est uniquement un fichier objet. Ses fonctions seront ensuite utilisables, rangées dans un module nommé `BinomialHeap`.

Les fichiers `test.ml` et `main.ml` utiliseront les fonctions du module `BinomialHeap`. Ils devront donc être compilés dans un deuxième temps :

```
$ ocamlc -c main.ml # Ceci produit main.cmo
$ ocamlc -o main binomialHeap.cmo main.cmo # Ceci lie les deux cmo dans un exécutable
↪ que l'on nomme "main"
$ ocamlc -c test.ml
$ ocamlc -o test binomialHeap.cmo test.cmo
```

Pour l'édition des liens, l'ordre des `cmo` est important : il doit respecter l'ordre de dépendance des modules entre eux.

Ceci produit des exécutables que vous pouvez ensuite lancer avec :

```
$ ./main
$ ./test</pre>
```

0.2 Arbres binomiaux

Pour tout entier naturel k , on définit par récurrence l'arbre binomial de degré k , que l'on note \mathcal{B}_k de la façon suivante :

- l'arbre \mathcal{B}_0 est l'arbre réduit à un seul noeud,
- l'arbre \mathcal{B}_k est l'arbre dont la racine a exactement pour fils les arbres $\mathcal{B}_{k-1}, \mathcal{B}_{k-2}, \dots, \mathcal{B}_0$ (dans cet ordre).

Question 0.2. Dessiner les premiers arbres binomiaux $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$.

Question 0.3. On considère deux copies \mathcal{T} et \mathcal{T}' d'un arbre binomial de degré k . Montrer que l'arbre obtenu en greffant \mathcal{T}' comme premier fils de la racine de \mathcal{T} est un arbre binomial de degré $k+1$ (c'est-à-dire que la racine de \mathcal{T}' devient un noeud fils de \mathcal{T}).

Question 0.4. Déterminer, en fonction de k , le nombre de noeuds d'un arbre binomial de degré k .

Dans la suite, on attribuera des étiquettes aux noeuds d'un arbre binomial. De plus, on aura besoin, étant donné un noeud quelconque, d'accéder rapidement d'une part à son noeud père et d'autre part à ses noeuds frères (c'est-à-dire la liste des noeuds situés après lui parmi les noeuds qui ont le même père). On utilisera donc le type :

```
type 'a binomial_node =  
  {  
    mutable label : 'a; (** Étiquette du noeud *)  
    mutable parent : 'a binomial_node; (** Pointeur vers le noeud parent *)  
    mutable children : 'a binomial_node list; (** Liste des enfants *)  
    mutable sibling : 'a binomial_node list; (** Liste des frères suivants *)  
    mutable degree : int; (** Degré de ce sous arbre *)  
  }  
  
and 'a binomial_tree = 'a binomial_node option
```

Question 0.5. Assurez-vous de bien comprendre la déclaration de type précédente. C'est crucial pour la suite. Posez des questions si besoin.

Question 0.6. Écrire une fonction `create_single` qui prend en paramètre une étiquette `x` et crée l'arbre binomial réduit à un seul noeud portant cette étiquette.

```
val create_single : 'a -> 'a binomial_node
```

0.3 Arbres en tas minimum

Définition 1. On rappelle qu'un arbre vérifie la propriété de *tas minimum* lorsque l'étiquette de chaque noeud est un minorant des étiquettes de tous les noeuds enfants.

Question 0.7. On suppose que l'on dispose d'un arbre en tas minimum, sauf pour un seul noeud qui est éventuellement plus petit que son parent. Écrire une fonction `retasse` qui prend en paramètre ce noeud et modifie l'arbre de sorte qu'à la fin de l'exécution cet arbre soit bien en tas minimum.

```
val retasse : 'a binomial_node -> unit
```

0.4 Tas binomiaux

0.4.1 Définition et opérations élémentaires

Un tas binomial est une liste d'arbres binomiaux qui possède les propriétés suivantes :

- pour chaque degré k , la liste contient au plus un arbre binomial d'ordre k ;

- la liste est triée par degrés croissants ;
- les arbres vérifient la propriété de tas minimum.

On représente un tas binomial par un élément de type :

```
type 'a binomial_heap = 'a binomial_node list
```

Ainsi un tas binomial est représenté par sa liste des noeuds racines.

Question 0.8. Écrire une fonction `minimum` qui renvoie le noeud d'étiquette minimale d'un tas binomial.

```
val minimum : 'a binomial_heap -> 'a binomial_node
```

Question 0.9. Quelle est la complexité de cette fonction en fonction du nombre total de noeuds contenus dans le tas binomial ? (Indication. Penser à la taille d'un arbre binomial et au fait que chaque degré apparaît au plus une fois.)

0.4.2 Réunion de deux tas binomiaux

L'opération primitive sur cette structure est la réunion, toutes les autres fonctions seront obtenues à partir de cette opération. La réunion de deux tas binomiaux construit un nouveau tas binomial contenant l'ensemble des noeuds.

Question 0.10. Écrire une fonction `link` qui prend en paramètres deux noeuds `y` et `z`. On suppose que ces deux noeuds sont des racines d'arbres binomiaux de degré $k - 1$. La fonction `link` choisit, entre `y` et `z`, le noeud d'étiquette minimale et lui ajoute comme nouveau fils l'autre noeud pour en faire la racine d'un arbre binomial de degré k .

Cette fonction ne vérifie aucune hypothèse sur les paramètres ou sur le résultat. Elle s'exécute en temps $O(1)$.

```
val link : 'a binomial_node -> 'a binomial_node -> 'a binomial_node
```

Pour réaliser l'union de deux tas binomiaux, on utilise un algorithme qui ressemble à l'algorithme d'addition de nombres posée avec retenues. On parcourt les listes des arbres des deux tas binomiaux et en gardant à chaque étape en mémoire une retenue (qui est aussi un arbre). À chaque étape, on regarde les deux arbres en tête de la liste des racines.

Quand il n'y a pas de retenue :

- s'ils sont de degrés différents, on place en tête du résultat l'arbre de plus petit degré ;
- s'ils sont de même degré, grâce à la fonction `link`, on les fusionne en un seul arbre de degré supérieur. Ce nouvel arbre est placé dans la retenue pour les tours suivants.

Quand il y a une retenue :

- si elle est de degré strictement inférieur aux deux arbres de tête, on la place en tête du résultat et on recommence sans retenue ;
- si les deux arbres de tête et la retenue ont le même degré, on place la retenue en tête du résultat et on lie les deux arbres de tête dans la nouvelle retenue ;
- si la retenue a le même degré qu'un seul des noeuds, on la lie avec ce noeud pour former la nouvelle retenue. Remarquer ce degré est nécessairement le minimum (pourquoi?).

Question 0.11. Convincez-vous de la correction de cet algorithme. Prouver qu'il termine.

Question 0.12. Écrire une fonction `union` qui prend en paramètres deux tas binomiaux et qui construit un tas binomial contenant l'union de tous les noeuds. Cette fonction modifie les structures passées en paramètres, donc après un appel à `union`, les paramètres ne désignent plus des tas binomiaux valides.

```
val union : 'a binomial_heap -> 'a binomial_heap -> 'a binomial_heap
```

0.4.3 Insertion et suppression

Pour chacune de ces fonctions, cherchez une méthode qui revient à faire des réunions de tas binomiaux bien choisis.

Question 0.13. Écrire une fonction `insert` qui prend en paramètres un tas binomial `heap` et une valeur `x` et qui modifie le tas afin d'insérer la valeur `x`.

```
val insert : 'a binomial_heap -> 'a -> 'a binomial_heap
```

Question 0.14. Écrire une fonction `extract_min` qui retire et renvoie la valeur minimale d'un tas binomial.

```
val extract_min : 'a binomial_heap -> ('a * 'a binomial_heap)
```

0.4.4 Modification de la clé

Question 0.15. Écrire une fonction `decrease_key` qui modifie un noeud en diminuant son étiquette. Il faudra penser à mettre à jour tout l'arbre binomial.

```
val decrease_key : 'a binomial_node -> 'a -> unit
```

0.4.5 Complexités

Question 0.16. Évaluer les complexités des opérations précédentes en fonction de la taille des tas binomiaux donnés en entrée. Justifier précisément la réponse.

0.5 Applications

0.5.1 Révisions

Question 0.17. Un algorithme célèbre vu en MP2I a besoin d'une structure de file de priorité. Lequel ? Déterminer sa complexité en général en fonction des opérations de base de la file de priorité, puis préciser ce que cela donne si on utilise spécifiquement des tas binomiaux.

0.5.2 S'asseoir loin

On dispose de N chaises sur une seule ligne et de K personnes ($K \leq N$) souhaitant s'asseoir sur ces chaises. Ces personnes se présentent une par une. Pour tenter de s'éloigner les unes des autres, elles choisissent la stratégie suivante : lorsque vient son tour, chaque personne s'assoit au milieu de la plus grande section de chaises vides contigües. Lorsqu'il y a plusieurs sections de taille maximale, on choisit la plus à gauche. Lorsque le milieu d'une section ne tombe pas exactement sur une chaise, la personne nous indique sa préférence personnelle, gauche ou droite, sur la chaise à choisir.

Une fois les personnes placées, on interroge votre programme avec Q requêtes. Chaque requête contient le numéro d'une chaise dans $\llbracket 0, N - 1 \rrbracket$. Vous devez répondre en indiquant le numéro de la personne assise sur cette chaise, ou -1 si personne ne se trouve sur la chaise.

On donne en entrée un fichier texte :

- la première ligne contient l'entier N suivi de l'entier K ;
- la deuxième ligne contient une liste de K caractères, L ou R, indiquant les préférences de placement de chaque personne (dans l'ordre des personnes) ;
- la troisième ligne contient l'entier Q ;
- puis chaque ligne suivante donne les requêtes auxquelles vous devez répondre.

En sortie, votre programme affiche la réponse à chaque requête, dans l'ordre d'entrée, chacune sur une ligne séparée.

Exemple. L'entrée :

```
3 3
RRL
3
1
3
2
```

produit la sortie :

```
2
3
1
```

Question 0.18. Écrire ce programme.

0.6 Point de vue du programmeur C

Considérer l'union comme une addition avec retenue relève plutôt du point de vue du programmeur OCaml récursif. Le programmeur C commencerait plutôt par fusionner les deux listes de racines en une liste ordonnée par degrés croissants, puis parcourerait avec une boucle cette nouvelle liste en fusionnant les arbres consécutifs de même degré. Programmez cette version, elle est intéressante.

TP 1

Jeu de Hex

1.1 Règles du jeu

On dispose d'une grille de cases hexagonales dont les deux côtés font le même nombre de cases. Au départ, aucune case n'est coloriée. Chaque joueur dispose d'une couleur et l'utilise pour colorier à chaque tour une case non encore coloriée.

Une position est gagnante pour le premier joueur s'il existe un chemin formé par des cases adjacentes coloriées avec sa couleur entre une case située sur la ligne du haut et une case située sur la ligne du bas (peu importe lesquelles).

De même, une position est gagnante pour le deuxième joueur si sa couleur permet de relier le côté de gauche au côté de droite.

1.2 Petits résultats théoriques

Question 1.1. Justifier qu'il n'y a pas de match nul, c'est-à-dire que si la grille est remplie alors il existe au moins un gagnant.

Question 1.2. On suppose qu'Alice a une stratégie ϕ gagnante depuis une position donnée P . Montrer que cette stratégie est gagnante si on part d'une position P' obtenue en reprenant les mêmes cases coloriées que dans P et en ajoutant une case coloriée par la couleur d'Alice.

On va démontrer par l'absurde que le premier joueur dispose d'une stratégie gagnante. On suppose pour cela que le deuxième joueur a une stratégie gagnante ϕ depuis n'importe quel premier coup du premier joueur.

Question 1.3. Alice joue un premier coup au hasard sur un premier plateau de Hex. En considérant un deuxième plateau de Hex, dans lequel Alice joue encore en premier un coup bien choisi et Bob joue en suivant ϕ , montrer qu'Alice peut imiter la stratégie ϕ sur le premier plateau. (On dit qu'Alice vole la stratégie de Bob.)

Question 1.4. Conclure. (En particulier, est-ce bien un raisonnement par l'absurde ?)

1.3 Programmer le jeu de Hex

Question 1.5. Proposer en C une structure de données `position_hex` permettant de représenter une position du jeu de Hex. Il faut notamment réfléchir à la manière dont vous représentez les cases du plateau. (Je vous conseille vivement de ne pas vous précipiter sur les questions suivantes tant que votre choix n'est pas validé par un informaticien diplômé.)

Question 1.6. Étant donné une case, écrire en C une fonction `cases_voisines` qui permet d'obtenir la liste des cases voisines. (La première difficulté de cette question vient du fait que la signature de la fonction n'est pas indiquée, en trouver qui sera pratique à manipuler est déjà une étape importante, qu'il peut encore être judicieux de faire valider par un informaticien chevronné.)

Question 1.7. Écrire une fonction `resultat_hex` qui prend en paramètre une position du jeu et qui renvoie un entier indiquant le résultat de cette position (victoire d’Alice, victoire de Bob ou bien partie pas encore terminée).

1.4 Élagage $\alpha\beta$

Le but est de programmer l’algorithme d’élagage $\alpha\beta$ avec une heuristique quelconque donnant le score d’un plateau.

Question 1.8. Commencer par programmer min-max et vérifier que cela marche sur des très petits plateaux. (On rappelle que le premier joueur a une stratégie gagnante depuis le plateau vide.)

Question 1.9. Quels paramètres faut-il ajouter pour modifier ce min-max en élagage $\alpha\beta$?

Question 1.10. Programmer l’élagage $\alpha\beta$.

Question 1.11. Soyez créatifs et proposez des heuristiques pour évaluer une position.

1.5 Variante du jeu

Une variante très classique cherche à compenser l’avantage systématique du premier joueur. Dans la variante, Alice joue sa première case. Bob peut alors décider soit de colorier une case (comme dans le jeu classique), soit de sacrifier son tour afin d’échanger les rôles. On garde alors telle quelle la première case coloriée, mais elle appartient désormais à Bob et c’est au tour d’Alice de colorier à nouveau une case. On joue ensuite normalement jusqu’à la fin du jeu.

TP 2

Manipulations autour des expressions régulières et des langages locaux

2.1 Langages locaux

Un langage local L est un langage décrit par un quadruplet :

- P , un ensemble de premières lettres autorisées en début d'un mot de L ,
- D , un ensemble de dernières lettres autorisées,
- F , un ensemble de facteurs de longueur 2 autorisés, tous les facteurs de longueur 2 d'un mot de L doivent être dans F ,
- α , un booléen qui indique si L contient le mot vide.

Dans ce sujet, on écrit quelques fonctions usuelles pour manipuler les langages locaux en C. On utilisera pour cela les types :

```
struct character_set_s {
    int length;
    unsigned char *members;
};
typedef struct character_set_s character_set;

struct language_s
{
    character_set first;
    character_set last;
    character_set allowed[256];
    bool has_empty;
};
typedef struct language_s language;
```

Le type `character_set` représente un ensemble de caractères. Ces caractères sont stockés *par ordre croissant* et *sans doublons* dans le tableau pointé par le champ `members`. La longueur de ce tableau est donnée par le champ `length`.

Le champ `allowed` du type `language` est un tableau qui à chaque caractère associe la liste des caractères suivants autorisés. Ceci permet de représenter la liste des facteurs de longueur 2 autorisés.

2.1.1 Reconnaissance d'un langage local

Question 2.1. Écrire une fonction `ll_belongs` qui détermine si un caractère appartient à l'ensemble décrit par un `character_set`.

```
bool ll_belongs(const character_set set, unsigned const char letter);
```

Question 2.2. Écrire une fonction `ll_match` qui détermine si un mot, donné par une chaîne de caractères `C`, appartient à un langage local donné.

```
bool ll_match(const language lang, unsigned const char* word);
```

2.1.2 Opérations sur les langages locaux

Question 2.3. On considère deux langages locaux L_1 et L_2 , respectivement sur deux alphabets A_1 et A_2 . Montrer que leur réunion n'est pas nécessairement un langage local, puis donner une condition suffisante sur A_1 et A_2 pour que $L_1 \cup L_2$ soit local.

Question 2.4. Écrire une fonction `ll_union` qui renvoie le langage local réunion des deux. Cette fonction ne vérifie pas que l'union est bien un langage local.

```
language ll_union(const language lang1, const language lang2);
```

Question 2.5. Montrer que la concaténation de deux langages locaux quelconques n'est en général pas un langage local. Donner une condition suffisante pour que ce soit le cas.

Question 2.6. Écrire une fonction `ll_concat` qui renvoie le langage concaténé de deux langages locaux.

```
language ll_contact(const language lang1, const language lang2);
```

Question 2.7. Si L est local, L^* est-il toujours local ?

Question 2.8. Écrire une fonction `ll_star` qui calcule l'étoile de Kleene d'un langage local.

```
language ll_star(const language lang);
```

2.2 Expressions régulières à la Kleene

Dans cette partie, nous définissons le type `kleene_regex` pour manipuler en C des expressions régulières à la Kleene.

On commence par les types de base :

```
enum kleene_regex_kind {
    KLEENE_REGEX_EMPTY = 0,
    KLEENE_REGEX_EPSILON,
    KLEENE_REGEX_WORD,
    KLEENE_REGEX_STAR,
    KLEENE_REGEX_UNION,
    KLEENE_REGEX_CONCAT,
};

struct kleene_regex_s {
    enum kleene_regex_kind kind;
};

typedef struct kleene_regex_s kleene_regex;
```

On définit ensuite une `struct` pour chaque type d'expression régulière. Chaque structure possède comme premier champ `base` de type `kleene_regex` :

```
struct kleene_regex_word_s {
    kleene_regex base;
    char *word;
};

typedef struct kleene_regex_word_s kleene_regex_word;

struct kleene_regex_binary_s {
```

```

        kleene_regex base;
        kleene_regex *left;
        kleene_regex *right;
};
typedef struct kleene_regex_binary_s kleene_regex_binary;
typedef struct kleene_regex_binary_s kleene_regex_union;
typedef struct kleene_regex_binary_s kleene_regex_concat;

struct kleene_regex_unary_s {
    kleene_regex base;
    kleene_regex *sub;
};
typedef struct kleene_regex_unary_s kleene_regex_star;

```

Le fait que base soit le premier champ permet de faire des conversions de type afin que toutes les fonctions puissent manipuler, en paramètres et en valeur de retour, des valeurs de type `kleene_regex` :

```

kleene_regex *kleene_empty(void);
kleene_regex *kleene_epsilon(void);
kleene_regex *kleene_word(const char *word);
kleene_regex *kleene_star(const kleene_regex *sub);
kleene_regex *kleene_union(const kleene_regex *left, const kleene_regex *right);
kleene_regex *kleene_concat(const kleene_regex *left, const kleene_regex *right);

```

Par exemple si on a construit une valeur de type `kleene_regex_union` :

```

kleene_regex_union *res_union = malloc(sizeof(kleene_regex_union));
res->base.kind = KLEENE_REGEX_UNION;

```

on peut renvoyer une valeur de type `kleene_regex` en convertissant explicitement :

```

return (kleene_regex *)res_union;

```

On fait l'hypothèse que, selon la valeur du champ `kind`, il est toujours possible de convertir une `kleene_regex` vers le type spécialisé. Par exemple, si `res->kind` vaut `KLEENE_REGEX_STAR` alors la valeur `res` doit permettre la conversion suivante :

```

kleene_regex_star *res_star = (kleene_regex_star *)res;

```

Question 2.9. Écrire les fonctions suivantes, qui permettent de construire des expressions régulières :

```

kleene_regex *kleene_empty(void);
kleene_regex *kleene_epsilon(void);
kleene_regex *kleene_word(const char *word);
kleene_regex *kleene_star(const kleene_regex *sub);
kleene_regex *kleene_union(const kleene_regex *left, const kleene_regex *right);
kleene_regex *kleene_concat(const kleene_regex *left, const kleene_regex *right);

```

Question 2.10. Écrire la fonction `kleene_is_empty` qui teste si une expression régulière engendre le langage vide.

```

bool kleene_is_empty(const kleene_regex *expr);

```

Question 2.11. Écrire la fonction `kleene_has_epsilon` qui teste si une expression régulière engendre un langage contenant le mot vide.

```

bool kleene_has_epsilon(const kleene_regex *expr);

```

Question 2.12. Écrire la fonction `kleene_first_letters` qui renvoie la liste des premières lettres des mots du langage engendré par une expression régulière.

```
char *kleene_first_letters(const kleene_regex *expr);
```

Question 2.13. Imaginez une manière d'écrire `kleene_match`, qui teste si une chaîne de caractères est dans le langage engendré par une expression. Aujourd'hui, on n'obtiendra probablement pas une fonction très efficace.

2.3 Expressions régulières étendues en C

Le standard POSIX permet de manipuler des expressions régulières, accessibles en incluant :

```
#include <regex.h>
```

Les expressions régulières sont de type `regex_t`, qui est un type opaque. Une expression régulière est fabriquée à partir d'une chaîne de caractères grâce à la fonction :

```
int regcomp(regex_t *restrict preg, const char *restrict regex, int cflags);
```

Le premier paramètre est initialisé et alloué par `regcomp`. Il stocke une représentation compilée de l'expression décrite par le paramètre `regex`. Le paramètre `cflags` permet de contrôler le comportement de l'expression régulière, mettez-le à 0 pour l'instant (et allez lire la page de manuel de `regcomp` pour découvrir les valeurs possibles). Cette fonction renvoie 0 lorsque l'expression a pu être compilée correctement.

```
Exemple. preg;
int rc = regcomp(&preg, "(sim[a-z]le) .* \\1", 0))) {
    printf("regcomp() failed, returning nonzero (%d)\n", rc);
    exit(EXIT_FAILURE);
}
```

Question 2.14. Pourquoi écrit-on `\\1` et pas `\1` dans l'expression précédente ?

La fonction `regexexec` permet ensuite de tester si une chaîne de caractères correspond à l'expression régulière. Cette fonction renvoie 0 lorsque la chaîne correspond.

```
int regexexec(
    const regex_t *restrict preg, /* expression compilée précédemment */
    const char *restrict string, /* chaîne à faire correspondre */
    size_t nmatch, /* Taille disponible pour le tableau pmatch */
    regmatch_t pmatch[restrict], /* Tableau dans laquelle regexexec stockera les positions
    ↪ où la regex s'applique */
    int eflags /* Drapeaux pour altérer le comportement, lisez le manuel */
);
```

Dans un premier temps, vous pouvez mettre la valeur `REG_NOSUB | REG_EXTENDED` en valeur pour `eflags`, respectivement afin :

- d'ignorer les paramètres `nmatch` et `pmatch`,
- d'utiliser les expressions régulières étendues.

```
Exemple. regexexec(&preg, "un simple motif vraiment assez simple", 0, NULL, 0)) {
    printf("Match failed\n");
}
```

N'oubliez pas de faire le ménage en mémoire quand vous n'avez plus besoin de l'expression régulière compilée :

```
void regfree(regex_t *preg);
```

2.3.1 Formats de dates

Un utilisateur peut entrer une date dans un champ. La plupart des développeurs imposent à l'utilisateur de saisir cette date dans un format fixé et refusent tout autre format en émettant un message d'erreur. Cependant, vous n'êtes pas ce type de développeur et vous voulez laisser aux utilisateurs le plus de choix de formats possible pour entrer une date.

Question 2.15. Écrivez une fonction qui prend une chaîne de caractères en entrée, essaie de l'interpréter comme une date et l'affiche en sortie au format ISO : YYYY-MM-DD, où YYYY donne les chiffres de l'année, MM ceux du mois et DD ceux du jour.

Question 2.16. Peut-on vérifier que la date est valide, au sens du calendrier, avec une expression régulière ?

2.3.2 Bannissez-les !

Le journal d'un serveur contient une ligne pour chaque requête qui lui est adressée. Trop d'attaquants tentent de s'introduire, j'aimerais bannir leurs adresses IP après une dizaine de tentatives d'attaques.

Voici un extrait du journal de mon serveur :

```
Sep 14 01:21:55 base-sujets sshd[67491]: Disconnected from authenticating user root
→ 41.94.88.46 port 41926 [preauth]
Sep 14 01:22:47 base-sujets sshd[67494]: Received disconnect from 92.255.85.69 port
→ 39260:11: Bye Bye [preauth]
Sep 14 01:22:47 base-sujets sshd[67494]: Disconnected from authenticating user root
→ 92.255.85.69 port 39260 [preauth]
Sep 14 01:23:05 base-sujets sshd[67496]: Invalid user jessie from 206.189.87.115 port
→ 39274
Sep 14 01:23:05 base-sujets sshd[67496]: Received disconnect from 206.189.87.115 port
→ 39274:11: Bye Bye [preauth]
Sep 14 01:23:05 base-sujets sshd[67496]: Disconnected from invalid user jessie
→ 206.189.87.115 port 39274 [preauth]
Sep 14 01:34:09 base-sujets sshd[67508]: Unable to negotiate with 80.76.51.43 port
→ 33490: no matching key exchange method found. Their offer: diffie-hellman-
→ group14-sha1,diffie-hellman-group-exchange-sha1,diffie-hellman-group1-sha1
→ [preauth]
Sep 14 01:34:24 base-sujets sshd[67510]: Unable to negotiate with 80.76.51.43 port
→ 48486: no matching key exchange method found. Their offer: diffie-hellman-
→ group14-sha1,diffie-hellman-group-exchange-sha1,diffie-hellman-group1-sha1
→ [preauth]
Sep 14 01:34:39 base-sujets sshd[67512]: Unable to negotiate with 80.76.51.43 port
→ 35102: no matching key exchange method found. Their offer: diffie-hellman-
→ group14-sha1,diffie-hellman-group-exchange-sha1,diffie-hellman-group1-sha1
→ [preauth]
```

Question 2.17. Extrairez ces adresses IP, en les affichant une par ligne. Chaque IP bannie doit n'apparaître qu'une seule fois dans le résultat.

Question 2.18. Limitez le blocage à 24h après la dernière tentative d'intrusion.

2.3.3 Ostracisme industrialisé

L'outil `fail2ban` lit les journaux d'activité d'un serveur à la recherche de tentatives d'intrusion, afin de bannir pendant un certain temps, directement au niveau de pare-feu, les adresses IP un peu trop insistantes. Pour détecter les comportements louches, il faut lire les messages d'erreurs d'une assez grande variété de services différents. Les filtres par défaut sont dans les sources : <https://github.com/fail2ban/fail2ban/tree/master/config/filter.d>

TP 3

Union-find

3.1 Définition et implémentation

On veut représenter des partitions d'un ensemble fini $\llbracket 0, n-1 \rrbracket$. Sur une telle partition, on veut pouvoir réaliser efficacement les deux opérations suivantes : déterminer si deux éléments sont dans la même classe de la partition, et modifier la partition pour réunir deux classes en une seule.

Pour chaque ensemble de la partition, on choisit arbitrairement un entier de cet ensemble qui joue le rôle de *représentant*.

La partition est représentée par un tableau `classes : int array` qui à chaque entier k associe :

- soit une valeur strictement négative, qui signifie que k est le représentant de sa classe ;
- soit une valeur positive u , ce qui signifie que pour trouver le représentant de k il faut déterminer le représentant de u .

Question 3.1. On considère le tableau `classes` suivant :

```
let classes = [| -1; -3; 1; 4; -2; 1; 2; 6 |]
```

Déterminer le nombre de classes de la partition et le représentant de chaque classe.

On veut écrire une fonction `find` qui à partir d'un entier v et du tableau des classes renvoie le représentant de la classe qui contient v .

Cette fonction utilise une optimisation, dite de *compression de chemins* : lors du premier appel à `find classes v`, il faut remonter d'entier en entier jusqu'à trouver le représentant u . On profite de cette remontée pour *modifier* le tableau `classes` en posant, une fois le représentant connu, `classes.(v) <- u`. Ainsi, l'appel suivant `find` pour l'entier v renverra (presque) directement la valeur u .

Avec l'exemple précédent, si l'on appelle `find classes 7`, alors le tableau vaudra à la fin de l'exécution :

```
[| -1; -3; 1; 4; -2; 1; 1; 1 |]
```

Question 3.2. Écrire la fonction `find : int array -> int -> int`.

Question 3.3. En déduire une fonction `meme_classe : int array -> int -> int -> bool` qui décide si deux entiers sont dans la même classe.

Étant donné deux entiers u et v , on veut à présent écrire une fonction `union classes u v` qui fusionne les deux classes contenant les entiers u et v . On commence par déterminer les représentants respectifs r_u et r_v de u et v et on pose au choix `classes.(ru) <- rv` ou bien `classes.(rv) <- ru`.

Pour choisir quelle modification apporter, on utilise une optimisation dite de *rang*. Le *rang* d'un représentant r est la valeur de `- classes.(r)`. Cette valeur est par définition positive. Il s'agit d'un majorant du nombre de sauts nécessaires pour trouver r à partir d'un entier quelconque de la classe.

Au début, tous les rangs valent 1. Lorsque l'on fusionne les classes associées à deux représentants r_u et r_v alors :

- on pose `classes.(rv) <- ru` lorsque le rang de r_u est strictement plus grand que celui de r_v ;
- on pose `classes.(ru) <- rv` lorsque le rang de r_v est strictement plus grand que celui de r_u ;
- en cas d'égalité de rang, on choisit arbitrairement `classes.(rv) <- ru` et on augmente le rang de r_u de 1 (ou le contraire, en échangeant `rv` et `ru`, c'est arbitraire).

Question 3.4. Écrire la fonction `union : int array -> int -> int -> unit`.

Question 3.5. Écrire une fonction `new_uf : int -> int array` qui initialise un tableau de classes où toutes les classes sont un singleton.

3.2 Étude de la complexité

On part d'une partition formée uniquement de singletons, dont les rangs initiaux valent 1. On réalise une succession de réunions.

Question 3.6. Montrer que, si k est un représentant de rang r_k , alors le nombre d'éléments de la classe de k est au moins 2^{r_k-1} .

Question 3.7. Montrer que `union` et `find` sont en $O(\log(n))$.

En réalité, ces opérations sont bien plus rapides, mais la preuve est plus compliquée.

3.3 Fabrication de labyrinthes

On dispose d'une grille rectangulaire de $n \times p$ cases carrées. Chaque côté d'une case est un mur, au début toutes les cases ont leurs quatre murs. On veut créer un labyrinthe aléatoirement tel que pour tout couple de cases, il existe un unique chemin reliant ces deux cases. Pour cela, on part de la liste des murs intérieurs (les murs sur le périmètre de la grille sont exclus), on mélange cette liste. On considère ensuite les murs mélangés un par un, et on abat un mur si et seulement s'il se trouve entre deux cases qui n'étaient pas reliées auparavant.

On numérote les cases de 0 à $np - 1$, ligne par ligne, en partant de la case en haut à gauche.

Question 3.8. Écrire une fonction `voisines : int -> int -> int -> int list` qui à partir des entiers n et p donnant la taille de la grille et d'un numéro de case k donne la liste des cases voisines.

Question 3.9. Écrire une fonction `murs : int -> int -> (int * int) list` qui donne la liste des murs de la grille. Chaque mur est représenté par le couple de cases qu'ils sépare.

Pour mélanger naïvement une liste d'éléments de type `'a list`, on utilise la méthode suivante : on attribue à chaque élément un entier au hasard pour former une liste de type `('a * int) list` puis on trie cette liste selon sa deuxième composante et enfin on projette le résultat sur la première composante.

OCaml dispose d'un générateur aléatoire dans le module `Random`. Ce générateur doit être initialisé avant sa première utilisation avec `Random.self_init ()`. On peut ensuite générer un entier aléatoire avec l'appel :

```
Random.int k
```

qui renvoie un entier de $\llbracket 0, k - 1 \rrbracket$.

Enfin, vous devez savoir écrire un tri efficace. Mais pour aujourd'hui vous pouvez utiliser :

```
List.sort : ('a -> 'a -> int) -> 'a list -> 'a list
```

Question 3.10. Écrire une fonction `melange_murs : (int * int) list -> (int * int) list`.

Question 3.11. Donner une condition suffisante sur les entiers générés aléatoirement pour que la liste mélangée soit obtenue uniformément parmi toutes les listes de murs.

Question 3.12. Générer un labyrinthe, représenté par la liste des murs *présents*.

Question 3.13. Ce serait quand même un labyrinthe beaucoup plus joli si les cases étaient hexagonales.

3.4 Faire la course

En OCaml, on peut mesurer le temps processeur utilisé par le programme actuel grâce à la fonction `Sys.time`, qui renvoie un flottant donnant le temps écoulé en secondes.

Question 3.14. Écrire un programme qui prend en paramètres deux entiers n_{min} et n_{max} . Pour chaque entier n entre ces deux bornes, il génère une instance d'unir et trouver de taille n , puis réalise aléatoirement n fusions (ce qui, parfois ne fait rien sur des sommets déjà fusionnés). En sortie, votre programme produit un fichier au format CSV qui donne le temps d'exécution (moyen) pour chaque valeur de n :

```
n,millis  
42,245  
43,257  
500,2104
```

Question 3.15. Tracer les courbes. Si vous êtes sages et efficaces, je vous montre gnuplot. Comparer les temps d'exécution en fonction de l'utilisation, ou non, des optimisations.

TP 4

Couplage maximum biparti

4.1 Introduction

On travaille dans ce sujet sur des *graphes bipartis*. Un graphe est dit biparti lorsque l'ensemble de ses sommets peut s'écrire sous la forme $V = L \cup R$, où $L \cap R = \emptyset$, et de telle sorte qu'il n'existe aucune arête entre deux sommets de L , ni aucune arête entre deux sommets quelconques de R .

On représentera les graphes par leur listes d'adjacence :

```
type graph = int list array
```

C'est un sujet où il est impératif de faire des dessins de graphes pour comprendre les définitions présentées.

4.2 Couplages

Un *couplage* est un ensemble d'arêtes M d'un graphe qui vérifie : si e_1 et e_2 sont deux arêtes de M , alors ces arêtes ne partagent aucun sommet commun. On dit que deux sommets du graphe sont couplés s'ils sont reliés par une arête de M .

Le but est de déterminer un couplage de cardinal maximum.

On représentera un couplage par un tableau de type :

```
type matching = int option array
```

Si `matching` est un couplage, alors `matching.(i)` vaut `None` si le sommet i n'est couplé avec aucun sommet. Sinon, `matching.(i)` indique le numéro j du sommet couplé avec i . Dans ce cas, on a nécessairement `matching.(j) = Some i`.

Question 4.1. Écrire une fonction `empty_matching` qui, à partir d'un graphe biparti, crée le couplage vide (c'est-à-dire le couplage où aucun sommet n'est couplé).

```
val empty_matching: graph -> matching
```

Question 4.2. Écrire une fonction `mate` telle que l'appel `mate matching v1 v2` ajoute l'arête entre v_1 et v_2 au couplage. Cette fonction n'effectue aucune vérification (notamment, elle ne vérifie pas si les deux sommets étaient déjà couplés, ni si l'arête existe effectivement dans le graphe).

```
val mate : matching -> int -> int -> unit
```

Question 4.3. Écrire une fonction `unmate` telle que l'appel `unmate matching v1` retire, si elle est présente dans le couplage, l'arête contenant v_1 . (On n'oubliera pas qu'il faut mettre à jour *deux* cases dans `matching` : celle de v_1 et celle de l'éventuel sommet couplé.)

```
val unmate : matching -> int -> unit
```

4.3 Sommets libres

Soit M un couplage d'un graphe biparti $G = (L \cup R, E)$. Un sommet *libre* est un sommet qui n'est couplé avec aucun sommet de G .

Question 4.4. Écrire une fonction `free_vertices` qui renvoie la liste des sommets libres du graphe.

```
val free_vertices : matching -> int list
```

4.4 Chemins améliorants

Soit M un couplage d'un graphe biparti $G = (L \cup R, E)$. Un *chemin améliorant* est une liste de sommets $[v_1; v_2; \dots; v_n]$ telle que :

- v_1 et v_n sont libres ;
- chaque arête $\{v_i, v_{i+1}\}$ est une arête de G ;
- les arêtes de la forme $\{v_{2i+1}, v_{2i+2}\}$ ne sont **pas** dans le couplage M ;
- les arêtes de la forme $\{v_{2i}, v_{2i+1}\}$ sont des arêtes de M .

Un chemin améliorant C peut se représenter en OCaml comme un couplage : il s'agit du couplage où l'on prend les arêtes $\{v_{2i+1}, v_{2i+2}\}$. Les autres arêtes du chemin améliorant se déduisent de M : le sommet qui suit chaque v_{2i} dans le chemin correspond au sommet couplé avec v_{2i} dans M .

Question 4.5. Écrire une fonction `symdiff` telle que l'appel `symdiff matching path` retire du couplage `matching` tous les sommets couplés dans `path`, puis ajoute dans `matching` les arêtes du couplage `path`.

```
val symdiff : matching -> matching -> unit
```

On peut déterminer un chemin améliorant grâce à un parcours en profondeur dans le graphe, mais en suivant systématiquement les arêtes présentes dans le couplage, en partant des sommets libres.

Question 4.6. Écrire une fonction `augmenting_path` telle que l'appel `augmenting_path g matching path` renvoie `true` exactement quand il existe un chemin améliorant dans le graphe g . Cette fonction stockera ce chemin dans le paramètre `path` (que l'on supposera égal au couplage vide initialement).

```
val augmenting_path : graph -> matching -> matching -> bool
```

Le parcours en profondeur pourra être par exemple programmé comme une fonction auxiliaire interne à `augmenting_path`, qui aurait le type :

```
val dfs : int -> int list -> bool
```

telle que l'appel `dfs v1 voisins` cherche un chemin améliorant passant par v_1 et renvoie un booléen indiquant si un tel chemin a été trouvé. Elle modifie le tableau `path` durant la construction du chemin améliorant.

4.5 Couplage maximum

Question 4.7. Programmer la fonction `best_matching`.

```
val best_matching : graph -> matching
```

4.6 Couplage parfait de poids maximum

Dans cette partie, on s'intéresse à des graphes bipartis ayant les propriétés suivantes :

- ils sont équilibrés, c'est-à-dire que les deux composantes de sommets ont le même cardinal,
- ils sont complets, c'est-à-dire qu'entre deux sommets d'une composante et de l'autre, l'arête est présente.

Un couplage est parfait lorsque tous les sommets sont couplés.

On attribue à chaque arête du graphe, non orienté, des poids positifs. Le poids d'un couplage est la somme des poids de ses arêtes. On cherche désormais à déterminer non plus un couplage de cardinal maximum mais un couplage parfait de poids maximum. On note $w(u, v)$ le poids de l'arête reliant u et v .

On décrit ici l'algorithme hongrois, qui permet de déterminer un couplage de poids maximum. Pour chaque sommet u , on attribue un *potentiel* $l(u)$. L'étiquetage est *valide* lorsque, pour tous sommets u, v , $l(u) + l(v) \geq w(u, v)$.

Question 4.8. Déterminer un étiquetage valide initial pour un graphe biparti.

La *marge* d'une arête est la quantité $l(u) + l(v) - w(u, v)$.

Le graphe d'égalité G_l est le sous graphe constitué uniquement des arêtes vérifiant $l(u) + l(v) = w(u, v)$. L'algorithme que l'on implémente recherche des chemins améliorants, tant que c'est possible, dans le graphe d'égalité et lorsque ce n'est plus possible, modifie les potentiels puis recommence l'étape précédente.

On modifie un peu l'algorithme de recherche d'un chemin améliorant : cette étape se fait avec un parcours qui crée des chaînes alternantes (entre arêtes non couplées et arêtes couplées). Ce parcours définit donc des *arbres alternants*. Un tel arbre a pour racine un sommet non couplé, chaque branche est un chemin alternant.

Si l'une des branches est un chemin améliorant, on interrompt la construction de l'arbre alternant et on améliore le couplage.

Si on ne trouve aucun chemin améliorant, on définit la *marge* δ de l'arbre comme étant la plus petite marge parmi celles de toutes ses arêtes. On modifie les potentiels en augmentant de δ tous ceux des nuds de la même composante bipartie que la racine, et en diminuant de δ ceux de l'autre composante.

Question 4.9. Montrer que l'arbre alternant reste dans le graphe d'égalité après cette opération. Montrer que la somme de tous les potentiels diminue.

Pour représenter l'arbre alternant, on pourra considérer un tableau qui à chaque nud associe le numéro du nud père dans l'arbre, un nud racine aura pour père -1 .

Question 4.10. Écrire une fonction qui détermine un couplage de poids maximum.

4.7 Un peu plus loin

Le sujet d'option informatique du concours commun Mines Ponts 2012 aborde les couplages de cardinal maximum. La troisième partie concerne l'algorithme hongrois dans sa version sans poids.

Comment pourriez-vous résoudre le problème 1045 (The Great Wall Game) de l'UVa ? <https://onlinejudge.org/external/10/1045.pdf>

TP 5

Petits jeux avec les processus légers

Processus légers en C

En C on dispose des fonctions suivantes dans `pthread.h` :

```
int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void *),
                  void *restrict arg);

int pthread_join(pthread_t thread, void **retval);
```

On dispose également de *mutex* fournis par la bibliothèque `pthread` :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex);
/* Section critique */
pthread_mutex_unlock(&mutex);
```

Les programmes utilisant la bibliothèque `pthread` doivent être compilés avec l'option `-pthread` de `gcc` :

```
gcc -pthread -o source.o source.c
gcc -pthread -o executable source1.o source2.o source3.o
```

Question 5.1. Écrivez un programme qui lance deux processus légers, chacun réalisant des `printf`. L'affichage est-il entrelacé ?

Question 5.2. Écrivez un programme qui lance plusieurs processus légers, qui incrémentent tous un même compteur partagé sans aucune précaution. Que constatez-vous ? Dans un deuxième temps, ajoutez des `printf` à chaque incrémentation et recommencez le test : avez-vous une explication ?

Question 5.3. Programmer l'incrémentation de compteur avec un mutex.

Des bulles

Le tri à bulles n'intéresse personne. Mais on peut imaginer une variante où on imagine qu'en chaque paire de cases consécutives du tableau à trier se trouve un lutin dont le travail est de permuter les deux cases lorsqu'elles ne sont pas ordonnées en croissant. Les lutins travaillent en parallèle.

Question 5.4. Les lutins doivent-ils se synchroniser ? Comment évaluer la terminaison ?

Question 5.5. Proposer un algorithme de tri à bulles par lutins parallèles.

Question 5.6. Programmer une fonction `tri_lutins` qui réalise ce tri. Est-ce rapide ?

Diviser pour régner, parallèlement

Le tri rapide se parallélise naturellement, étant donné que les appels récursifs sur chaque sous liste sont totalement indépendants.

Question 5.7. Écrire une fonction `tri_rapide_parallele_naif` qui effectue le tri rapide d'un tableau. Au lieu de faire un simple appel récursif, elle démarre un nouveau processus léger pour chaque traitement récursif de la sous liste.

Créer un processus léger demande un peu de temps de traitement, l'opération n'est pas si légère que cela. Il n'est pas vraiment rentable de séparer le traitement des sous listes lorsque leur taille est vraiment trop petite. De plus, le nombre de processus créés par la fonction précédente dépasse en général largement le nombre de processeurs présents sur la machine : ceci empêche d'obtenir un vrai gain en parallélisant.

Supposons que l'on dispose de p processeurs. Nous allons plutôt écrire une version qui se limite à créer p processus légers.

La phase de partition autour d'un pivot peut être parallélisée, en créant p processus légers. Le tableau de départ est découpé en p morceaux de tailles équivalentes. On choisit un pivot, qui est partagé par tous les processus légers. Chaque processus léger est chargé de découper le tableau en deux parties autour du pivot. On regroupe alors tous les éléments inférieurs au pivot d'une part, tous les éléments supérieurs d'autre part. Le résultat ne sera pas stocké sur place.

Les processus se divisent alors en deux groupes de $p/2$ processeurs et on recommence l'opération.

Question 5.8. Écrire une fonction `tri_rapide_parallele` qui effectue ce tri. Le nombre p pourra être donné en paramètre.

File d'attente distribuée

Dans cette partie, on propose de créer une structure de données **deque** (*double ended queue*). Il s'agit d'un mélange entre une file et une pile : on peut réaliser des suppressions de chaque côté avec à peu près la même performance et des ajouts uniquement à la fin de la structure. Cette structure devra de plus être *thread-safe*, c'est-à-dire que son comportement est correct lorsqu'elle est utilisée simultanément par plusieurs fils d'exécution.

Les données stockées peuvent être quelconques, on stocke un tableau de pointeurs vers ces données, et deux indices `top` et `bottom`. Les éléments de la deque sont les éléments entre les indices `top` et `bottom`. L'indice `top` donne la tête de la deque, l'indice `bottom` donne la queue de la deque. On se donne également un mutex pour bloquer l'accès à `top`.

```
struct deque_s {
    void **data;
    int capacity;
    int top;
    int bottom;
    pthread_mutex_t top_lock;
};
typedef struct deque_s deque;
```

Question 5.9. Écrire une fonction `deque_new` qui crée une deque et renvoie un pointeur vers cette deque. On choisira une taille arbitraire en capacité initiale.

Le retrait d'un élément en tête se fait en incrémentant `top`, le retrait en queue se fait en décrémentant `bottom`. L'ajout d'un élément en queue se fait en plaçant un élément en position `bottom+1` et en incrémentant `bottom`. On a déjà vu que l'on pouvait raisonner avec des indices modulo la capacité, afin de réutiliser les cases du début du tableau lorsque l'on arrive à la fin.

Question 5.10. Écrire une fonction `deque_is_empty` qui teste si une deque est vide. Cette fonction doit être *thread-safe*, sans utiliser de mutex, donc il faut réfléchir à l'ordre dans lequel vous faites vos opérations.

Cependant, il peut arriver que toutes les cases du tableau soient effectivement utilisées. Dans ce cas, comme pour les tableaux dynamiques, on alloue un tableau de taille plus grande (d'un rapport géométrique par rapport à la capacité précédente) et on recopie les éléments dans ce tableau.

Question 5.11. Écrire une fonction `deque_resize` qui agrandit le tableau `data`.

Question 5.12. Écrire une fonction `deque_poptop` qui renvoie l'élément de tête et le supprime de la deque. Cette fonction renvoie `NULL` si la deque est vide. Cette fonction doit être *thread-safe*, notamment pour l'accès à `top`.

Question 5.13. Écrire une fonction `deque_pushbottom` qui ajoute un élément en fin de deque, si nécessaire en la redimensionnant.

Question 5.14. Écrire une fonction `deque_popbottom` qui supprime un élément en fin de deque, en renvoyant `NULL` si la deque est vide.

Vol de travail

La structure de deque permet de réaliser du vol de travail entre plusieurs processus légers.

Par exemple dans le cas du tri rapide précédent, lorsque l'on choisit un mauvais pivot, la charge de travail est déséquilibrée entre les deux groupes de processus qui traitent chaque partie. Les travailleurs ayant la tranche la plus courte auront terminé leur calcul bien en avance.

Ces processus peuvent alors voler du travail aux autres processus. À chaque instant, un processus a une liste de tâches à exécuter, stockée dans sa propre deque. Il retire les éléments de la deque un par un en queue. L'ajout de travail se fait en fin de la deque.

Lorsque la deque d'un processus est vide, il tente de voler du travail, par le *haut* de la deque, à un autre processus.

On va supposer dans cette partie que les éléments de la deque sont des pointeurs vers des fonctions de type :

```
void task();
```

L'ensemble de toutes les deques est stockée dans une structure :

```
struct worker_queues_s {
    int nb_workers;
    deque *queue;
};
typedef struct worker_queues_s worker_queues;
```

Chaque travailleur a un identifiant, qui lui permet d'accéder à sa propre deque :

```
struct worker_s {
    worker_queues workers;
    int id;
};
typedef struct worker_s worker;
```

Question 5.15. Écrire une fonction `work_stealing_thread` qui exécute un processus léger, prenant en paramètre un pointeur qui peut être converti vers le type `worker *`, qui réalise les tâches de sa deque tant qu'elle n'est pas vide, puis vole du travail aux autres. Cette fonction n'est pas censée se terminer, mais on pourra appeler à des moments bien choisis la fonction `sched_yield`, qui ne prend aucun paramètre et indique à l'ordonnanceur que le thread n'a rien à faire et qu'il est possible (mais pas obligatoire) de laisser la place à un autre thread.

Cette fonction est dans `sched.h`.

Question 5.16. Écrire le tri rapide parallélisé voleur.

TP 6

Résolution de SAT

Ce sujet étudie la résolution de SAT pour des formules en forme normale conjonctive, en utilisant certaines heuristiques pour accélérer si possible la résolution.

6.1 Structures de données

On s'intéresse à des formules logiques en forme normale conjonctive. On note \mathcal{V} l'ensemble des variables. On représente un littéral en OCaml par :

```
type littéral =  
  | X of int  
  | NonX of int
```

Les formules logiques en forme normale conjonctive sont représentées par le type :

```
type fnc = littéral list list
```

Une forme normale conjonctive est donnée par une liste de clauses, et chaque clause par la liste de ses littéraux.

6.2 Simplification élémentaire

Si un littéral apparaît sous la forme x_i et \bar{x}_i dans une clause, alors on peut simplifier la clause en la retirant de la formule. De même, si un littéral apparaît plusieurs fois dans une même clause, on peut simplifier la clause en ne gardant qu'une seule occurrence de ce littéral.

Question 6.1. Écrire une fonction `simplifie_antagonistes` qui renvoie une forme normale conjonctive après application de ces simplifications.

6.3 Règle de propagation unitaire

Nous supposons que nous disposons d'une formule F_{initial} en forme normale conjonctive.

Soit F une formule en forme normale conjonctive. On suppose que F contient une clause unitaire réduite au littéral l . Un tel littéral est appelé *littéral isolé*. Nous pouvons alors simplifier la formule F de la manière suivante :

- en supprimant toutes clauses de F qui contiennent l ,
- en supprimant $\neg l$ de toutes les clauses de F (ainsi si une clause ne contient que $\neg l$ elle est remplacée par la clause vide \perp).

Par exemple la formule $F = (\neg x_{(0,2)}^3) \wedge (\neg x_{(0,2)}^3 \vee x_{(2,5)}^6) \wedge (x_{(0,2)}^3 \vee \neg x_{(1,4)}^7)$ contient un unique littéral isolé $\neg x_{(0,2)}^3$. En simplifiant F par ce littéral, on obtient la formule $F' = \neg x_{(1,4)}^7$.

On justifie formellement cette simplification. Soit F une formule en forme normale conjonctive contenant un littéral isolé l (associé à la variable propositionnelle p : on a donc $l = p$ ou $l = \neg p$). F peut s'écrire alors sous la forme $F = l \wedge F_1 \wedge F_2 \wedge F_3$, où :

- F_1 est une formule constituée de clauses contenant chacune le littéral l ;
- F_2 est une formule constituée de clauses contenant chacune le littéral $\neg l$ et ne contenant pas le littéral l ;
- F_3 est une formule constituée de clauses ne contenant chacune ni le littéral l , ni le littéral $\neg l$.

Remarquons que chacune des formules F_1, F_2 et F_3 peut être réduite à la formule vide, satisfiable.

La formule simplifiée de F par le littéral l est alors la formule $F' = F_2' \wedge F_3$, où F_2' s'obtient à partir de F_2 en supprimant toutes les occurrences du littéral $\neg l$ dans chacune des clauses de F_2 .

Question 6.2. Soit σ une valuation de $\mathcal{V} \setminus \{p\}$. Montrer que σ satisfait F' si et seulement s'il existe une valuation $\bar{\sigma}$ de \mathcal{V} prenant les mêmes valeurs que σ sur $\mathcal{V} \setminus \{p\}$ qui satisfait F . On précisera la valeur de $\bar{\sigma}(p)$.

Question 6.3. Appliquer l'algorithme de propagation unitaire à la formule

$$F = x_{(0,0)}^1 \wedge (x_{(2,2)}^4 \vee x_{(3,6)}^6 \vee x_{(7,7)}^7) \wedge (\neg x_{(0,0)}^1 \vee \neg x_{(3,6)}^6)$$

Question 6.4. Écrire une fonction `nouveau_lit_isole` qui, à partir d'une formule F , renvoie un littéral isolé de F . Si F ne contient pas de tel littéral, la fonction renverra le littéral x_{-1} qui n'est pas utilisé comme variable propositionnelle par ailleurs.

Question 6.5. Écrire une fonction `simplification` qui, à partir d'un littéral l et d'une formule F , renvoie la formule simplifiée F' après propagation du littéral l dans F .

Question 6.6. Écrire une fonction `propagation` qui effectue cette simplification sur une formule, tant qu'elle est possible.

Question 6.7. On appelle *taille* d'une formule F le nombre total de littéraux apparaissant dans ses clauses. Évaluer le nombre d'opérations effectuées par les fonctions `nouveau_lit_isole`, `simplification`, puis par la fonction `propagation` quand elles s'appliquent à une formule de taille n .

6.4 Règle du littéral infructueux

On décrit maintenant une autre méthode de déduction plus puissante combinant la propagation unitaire et une opération appelée *règle du littéral infructueux* décrite ci-dessous.

Étant donné une formule F en forme normale conjonctive et une variable propositionnelle x :

- si l'algorithme de propagation unitaire appliqué à la formule $F \wedge \neg x$ permet de déduire la clause vide alors on ajoute la clause x à F ;
- si l'algorithme de propagation unitaire appliqué à la formule $F \wedge x$ permet de déduire la clause vide alors on ajoute la clause $\neg x$ à F .

Question 6.8. Justifier formellement que si l'on peut déduire la clause vide à partir de la formule $F \wedge \neg x$ alors $F \equiv F \wedge x$.

Question 6.9. Écrire une fonction `variables` qui, à partir d'une formule, renvoie la liste de ses variables sans doublons.

On rappelle que la fonction `List.flatten` permet de concaténer tous les éléments d'une liste de listes.

Question 6.10. Écrire une fonction `deduction` qui, à partir d'un tableau, d'une variable x et d'une formule F , renvoie 1 si la règle du littéral infructueux permet d'ajouter la clause x à F , -1 si elle permet d'ajouter la clause $\neg x$ à F et 0 sinon.

Nous proposons un deuxième algorithme de propagation basé sur la règle du littéral infructueux. Celui-ci consiste à appliquer la propagation unitaire et, quand celle-ci ne permet plus de déduire de nouvelles clauses, à appliquer la règle du littéral infructueux pour obtenir une nouvelle clause unitaire de la forme x ou $\neg x$. Dès lors on peut reprendre la propagation unitaire. Le processus s'arrête lorsque ni la propagation unitaire ni la règle du littéral infructueux ne permettent de déduire de nouvelles clauses.

Question 6.11. Écrire une fonction `propagation2` qui met en uvre cet algorithme.

6.5 Résoudre SAT

Question 6.12. Écrire une fonction qui résout SAT pour les formes normales conjonctives, la plus efficace que vous pourrez.

6.6 Transition de phase

Dans cette partie, on fixe arbitrairement un nombre N de variables propositionnelles. On s'intéresse à une restriction du problème de la satisfiabilité, que l'on note $n, p - SAT$, où n et p sont des entiers.

On ne considère que la satisfiabilité de formules de la forme :

$$\bigwedge_{i=1}^n \bigvee_{j=1}^p a_{i,j}$$

où les $a_{i,j}$ sont des littéraux.

On cherche à estimer la probabilité qu'une telle formule soit satisfiable en fonction du paramètre $\rho = n/p$. On va donc générer au hasard des formules et compter la proportion de formules satisfiables parmi elles.

On utilisera pour cela en OCaml les fonctions du module `Random` :

```
Random.init : int -> unit
Random.int : int -> int
```

La fonction `Random.init` sert à initialiser la graine du générateur de nombres aléatoires (une même graine générera la même suite de nombre aléatoires : dans des applications réelles, il est courant d'initialiser cette graine avec une donnée qui varie, par exemple la date ou des données provenant du pool d'entropie de l'ordinateur, telles que l'activité réseau, clavier, souris, etc., récente). Cette fonction doit être appelée une seule fois avant de générer le moindre nombre aléatoire.

Par la suite, chaque appel à `Random.int k` renvoie un entier tiré uniformément dans $\llbracket 0, k - 1 \rrbracket$.

Question 6.13. Écrire une fonction `random_litteral : int -> litteral` qui prend en paramètre un entier N et choisit au hasard uniformément un littéral parmi $x_0, x_1, \dots, x_{N-1}, \neg x_0, \dots, \neg x_{N-1}$.

Question 6.14. Écrire une fonction `random_clause : int -> int -> litteral list` qui prend en argument deux entiers N et p et construit au hasard uniformément une clause de la forme $\bigvee_{j=1}^p a_j$, où les a_j sont des littéraux pris parmi N variables.

Question 6.15. Écrire une fonction `random_instance : int -> int -> int -> fnc` qui prend en arguments trois entiers N , n et p et construit une instance de $n, p - SAT$ tirée au hasard avec au plus N variables.

Question 6.16. Écrire une fonction `proportion_npsat : int -> int -> int -> float` qui prend en arguments trois entiers N , n et p , qui génère un nombre raisonnable de votre choix d'instances de $n, p - SAT$ et qui renvoie la proportion d'instances satisfiables.

Question 6.17. Écrire une fonction `proportion_sat : int -> float -> float` qui prend en arguments un entier N et un flottant ρ et qui estime la probabilité qu'une formule de $n, p - SAT$ soit satisfiable lorsque les entiers n et p vérifient $\rho = n/p$.

Question 6.18. Faites varier ρ pour N fixé. Constatez-vous quelque chose ?

TP 7

Commutation d'interrupteurs

Nous considérons un système commandé par un tableau de bord comportant N interrupteurs, chacun pouvant être baissé ou levé. On désire tester ce système (pour le valider ou pour effectuer une opération de maintenance) en essayant mécaniquement chacune des 2^N configurations possibles pour l'ensemble des interrupteurs. Le coût de cette opération, qu'elle soit réalisée par un opérateur humain ou par un robot, sera le nombre total de mouvements d'interrupteurs nécessaires. Nous supposons que chaque fois qu'un interrupteur est commuté le système effectue un diagnostic automatiquement et instantanément. Les interrupteurs sont indexés de 0 à $N - 1$.

7.1 Parties d'un ensemble

Nous appelons *partie* un sous-ensemble fini de l'ensemble \mathbb{N} des entiers naturels. Un élément d'une partie est appelé *indice*. La *différence symétrique* de deux parties P et Q est définie par :

$$P \Delta Q = (P \setminus Q) \cup (Q \setminus P) = (P \cup Q) \setminus (P \cap Q)$$

Question 7.1. Montrer, de la façon la plus rapide possible, que la différence symétrique est commutative et associative.

Pour tout entier n positif ou nul, nous notons $\mathcal{I}_n = \{0, \dots, n - 1\}$ l'ensemble des entiers inférieurs strictement à n et \mathcal{P}_n l'ensemble des parties de \mathcal{I}_n .

Une partie sera représentée par une liste chaînée d'indices distincts **apparaissant dans l'ordre croissant des entiers**. On utilisera les types suivants :

```
type indice = int
type partie = indice list
```

Question 7.2. Écrivez la fonction `cardinal` qui renvoie le nombre d'éléments d'une partie. Vous ne ferez bien entendu pas appel à la fonction de bibliothèque `List.length`.

```
val cardinal : partie -> int
```

Question 7.3. Écrivez la fonction `delta` qui réalise la différence symétrique de deux parties. Le nombre d'opérations ne devra pas excéder $O(m + n)$ où n et m sont les cardinaux des éléments. Nous rappelons que dans toute liste chaînée représentant une partie les indices sont distincts et doivent apparaître dans l'ordre croissant des entiers.

```
val delta : partie -> partie -> partie
```

Nous représentons une configuration des interrupteurs par la partie formée des indices des interrupteurs baissés.

Question 7.4. Écrivez une fonction `test` qui imprime la liste des indices d'interrupteurs à commuter pour passer d'une configuration à une autre.

```
val test : partie -> partie -> unit
```

7.1.1 Énumération des parties par incrément

À toute partie P , nous associons l'entier $e(P) = \sum_{i \in P} 2^i$ (avec la convention $e(\emptyset) = 0$). Nous définissons le successeur de P comme l'unique partie Q telle que $e(Q) = e(P) + 1$.

Question 7.5. Écrivez une fonction `successeur : partie -> partie` qui renvoie le successeur d'une partie. Le nombre d'opérations ne devra pas excéder $O(l)$ où l est le plus petit indice absent dans la partie donnée en argument.

```
val successeur : partie -> partie
```

En application de ce mode d'énumération des parties, nous voulons réaliser le test de toutes les configurations d'interrupteurs. **Au début et à la fin du test tous les interrupteurs seront levés.**

Question 7.6. Écrivez un programme qui imprime la liste des indices des interrupteurs à commuter pour réaliser la totalité du test pour N interrupteurs et qui examine les configurations dans l'ordre défini par le successeur. L'argument de cette fonction sera l'entier N .

```
val test_incr : int -> unit
```

Question 7.7. Quel est le coût, dans le pire cas, du passage d'une configuration à la suivante ? Quel est le coût amorti si on énumère toutes les configurations ?

7.2 Énumération des parties par un code de Gray

Nous notons $\langle u_0, \dots, u_{l-1} \rangle$ une suite finie de l entiers. La concaténation de deux suites finies de longueur l et l' respectivement est une suite finie de longueur $l + l'$ définie par :

$$\langle u_0, \dots, u_{l-1} \rangle \cdot \langle u'_0, \dots, u'_{l'-1} \rangle = \langle u_0, \dots, u_{l-1}, u'_0, \dots, u'_{l'-1} \rangle$$

La suite vide, notée ε est de longueur 0. Une suite finie U est *préfixe* d'une autre suite finie V s'il existe une suite finie W telle que $V = U \cdot W$ (autrement dit U est le début de V).

Pour tout entier n positif ou nul, nous considérons la suite finie $T(n)$ de longueur $2^n - 1$ définie par :

$$T(0) = \varepsilon \text{ et } T(n+1) = T(n) \cdot \langle n \rangle \cdot T(n)$$

Pour tout entier i positif ou nul, nous notons t_i le $(i+1)$ -ème élément de $T(n)$ s'il existe. Puisque $T(n)$ est préfixe de $T(n+1)$, la suite $(t_i)_{i \geq 0}$ est définie sans ambiguïté. Enfin, nous posons $S_0 = \emptyset$ et pour tout entier i positif ou nul nous définissons l'ensemble $S_{i+1} = S_i \Delta \{t_i\}$.

Question 7.8. Donnez les valeurs de $T(1)$, $T(2)$, $T(3)$ et $T(4)$. Donnez la valeur de S_i pour tout i inférieur ou égal à 15.

Nous voulons montrer que les S_i peuvent être utilisés pour énumérer les parties de \mathcal{I}_n , et ainsi résoudre notre problème d'interrupteurs.

Question 7.9. Donnez la valeur de S_{2^n-1} pour tout $n \geq 0$.

Question 7.10. Montrez que pour tout $n > 0$ et tout $i < 2^n$, on a $S_{2^n+i} = S_i \Delta \{n-1, n\}$.

Question 7.11. En déduire que pour tout $n \geq 0$ l'ensemble $\mathcal{P}_n = \{S_0, S_1, \dots, S_{2^n-1}\}$.

Nous allons utiliser ce résultat pour résoudre le problème des interrupteurs. Comme dans la deuxième partie, nous imposons que les interrupteurs soient levés au début et à la fin du test.

Question 7.12. Écrivez une fonction `test_gray : int -> unit` s'inspirant des résultats de cette partie, qui imprime une liste d'indices d'interrupteurs à commuter pour réaliser la totalité du test. L'argument de cette fonction sera le nombre d'interrupteurs N .

Question 7.13. Quel est le coût du test avec cette méthode (c'est-à-dire le nombre total d'interrupteurs à commuter) ? Peut-on réaliser le test à un coût moindre ?

Question 7.14. On s'intéresse enfin à la notion de *successeur de Gray* d'une partie. Donnez une expression de t_i en fonction de S_i pour i impair. Déduisez-en la fonction `gray : partie -> partie` qui prend en argument une partie et renvoie celle qui la suit immédiatement dans l'ordre défini par la suite $(S_i)_{i \geq 0}$.

7.3 Sac à dos

On dispose d'un sac à dos qui permet de transporter un volume V et dont la charge pondérale maximale est P . On possède n objets, numérotés de 0 à N . Deux fonctions `volume` et `poids` que l'on supposera prédéfinies en OCaml donnent respectivement le volume $v(i)$ et le poids $p(i)$ de l'objet i .

On veut transporter dans le sac le sous-ensemble de ces N objets qui représente le volume maximal inférieur ou égal à V (perte minimale d'espace), et à volume égal le sous-ensemble qui correspond au poids maximal inférieur ou égal à P . On qualifiera d'optimal un tel sous-ensemble.

On supposera que $\forall i \in \llbracket 0, N-1 \rrbracket, v(i) \leq V$ et $p(i) \leq P$. Ceci garantit l'existence d'au moins une solution. On remarquera qu'il peut y avoir plusieurs solutions optimales équivalentes. Dans la suite du problème on ne s'intéressera qu'à la recherche de l'une de ces solutions.

Le problème du sac à dos est NP-difficile. Pour la recherche de l'une des solutions optimales de la résolution du problème du sac à dos, on envisagera tous les sous-ensembles possibles de l'ensemble des N objets.

Question 7.15. Énoncer le problème de décision associé. Montrer qu'il est NP-complet.

7.3.1 Utilisation du calcul des parties d'un ensemble

On définit :

$$\mathcal{P}_1(\llbracket 0, N-1 \rrbracket) = \left\{ (e, \sum_{i \in e} p(i), \sum_{i \in e} v(i)) \mid e \in \mathcal{P}(\llbracket 0, N-1 \rrbracket) \right\}$$

Question 7.16. En s'inspirant de la fonction `gray` précédente, écrire une fonction `chargements` qui prend en argument un entier N et un triplet de l'ensemble $\mathcal{P}_1(\llbracket 0, N-1 \rrbracket)^*$, et qui renvoie le triplet suivant, selon l'ordre de votre choix.

Question 7.17. Écrire une fonction `sac_a_dos` telle que l'appel `sac_a_dos p v n` renvoie le triplet qui correspond à l'une des solutions optimales.

```
val sac_a_dos : int -> int -> int -> int list * int * int
```

7.3.2 Une stratégie alternative d'énumération

Pour engendrer toutes les parties non vides de l'intervalle $\llbracket 0, N-1 \rrbracket$, il existe une autre solution. On considère que les listes sont triées dans l'ordre **décroissant**. La méthode consiste à calculer les $2^N - 2$ termes de la suite définie par :

- $u_0 = [0]$;
- si $0 < p < 2^N - 1$ et si $u_{p-1} = [x_0; x_1; \dots; x_{k-1}]$ alors :
 - si $k > 1$ et $x_0 < N-1$, alors $u_p = [x_0 + 1; x_0; x_1; \dots; x_{k-1}]$;
 - si $k > 1$ et $x_0 = N-1$, alors $u_p = [x_1 + 1; x_2; \dots; x_{k-1}]$;
 - si $k = 1$, u_p n'est défini que si $x_0 < N-1$ par $u_p = [x_0 + 1; x_0]$.

Question 7.18. Calculer manuellement la suite u_0, \dots, u_6 pour $N = 3$. Discuter l'intérêt de cette énumération par rapport à la précédente.

Question 7.19. Écrire une fonction `terme_suivant` qui calcule u_p à partir de N et u_{p-1} en temps constant.

```
val terme_suivant : int -> int list -> int list
```

Question 7.20. Écrire une fonction `sac_a_dos_alt` qui calcule une solution optimale pour n objets.

```
val sac_a_dos_alt : int -> int -> int -> int list * int * int
```

7.3.3 Un calcul de complexité

Question 7.21. Calculer la complexité $T(n)$ de la fonction `sac_a_dos_alt` en comptant uniquement le nombre d'utilisations de l'opérateur `::`.

Question 7.22. Rappeler la complexité de `1 @ 1'`.

Question 7.23. Calculer en fonction de $n = |e|$ la complexité de la fonction `parties` en prenant uniquement en compte l'opérateur de liste `::`.

Question 7.24. Comparer les deux complexités calculées.

TP 8

Approximation commerciale

8.1 Introduction

Dans ce sujet, inspiré du sujet Mines 2008, on résout de façon approchée (mais sans garantie de précision : pourquoi ?) puis exacte le problème du voyageur de commerce.

Un cycle hamiltonien dans un graphe est un cycle qui passe exactement une fois par chaque sommet du graphe. La recherche d'un cycle hamiltonien dans un graphe quelconque est un problème NP-complet.

Le problème du voyageur de commerce consiste à trouver un cycle hamiltonien dans un graphe complet non orienté pondéré, tel que le poids du cycle est minimal.

Question 8.1. Formuler le problème de décision associé au problème du voyageur de commerce. Méditer son caractère NP-complet, en prolongeant la méditation chez soi.

Ce sujet est accompagné d'une archive `tsp.tar.gz` contenant un squelette de code à compléter.

8.2 Représentation des graphes et des tours

On manipule des graphes complets pondérés dont les sommets sont numérotés par les entiers à partir de 0. On stocke les poids des arêtes dans un tableau à une seule dimension. Si n est le nombre de sommets du graphe, le poids de l'arête entre le sommet u et le sommet v est en case $un + v$. On trouve la même valeur en case $vn + u$.

Un cycle hamiltonien sera dans la suite appelé simplement un *tour* du graphe. On représente un tour par un tableau de sommets qui donne dans l'ordre les sommets du tour. En fonction du sommet de départ choisi, il y a donc plusieurs représentations d'un même tour.

Question 8.2. Écrire la fonction `poids_tour` calcule le poids d'un tour, le poids étant défini comme la somme des poids des arêtes composant le cycle.

8.3 Heuristique du plus proche voisin

On construit dans cette partie un tour en choisissant un sommet de départ, puis à chaque étape on prend comme sommet suivant le sommet le plus proche du sommet actuel parmi ceux qui ne sont pas encore sur le tour.

Question 8.3. Écrire la fonction `plus_proche` qui détermine le sommet le plus proche d'un sommet donné, uniquement parmi ceux qui ne sont pas déjà dans le tour.

Question 8.4. Écrire la fonction `tour_plus_proche` qui renvoie le tour obtenu par l'heuristique du plus proche voisin, à partir d'un sommet de départ donné.

Question 8.5. Le résultat dépend-il du sommet de départ choisi ?

Question 8.6. Démontrer que cette méthode ne renvoie (évidemment) pas toujours un tour de poids minimal. Un graphe à 4 sommets suffit.

8.4 Amélioration itérative

Dans cette partie, on part d'un tour déjà existant auquel on applique une transformation appelée *2-opt*, tant que cela améliore le tour.

Soit T un tour, décrit par le tableau des sommets visités. Soient i et j deux entiers vérifiant :

- $0 \leq i \leq n - 3$,
- $i + 2 \leq j \leq n - 1$,
- $(i, j) \neq (0, n - 1)$.

Dit autrement, i et j sont les positions de deux sommets du tour, séparés par au moins un autre sommet sur le tour.

Soit $i' = i + 1$ et $j' = (j + 1) \bmod n$. Le tour $2\text{-opt}(T, i, j)$ est le tour dont les arêtes sont obtenues à partir de celles de T :

- en retirant de T les arêtes $\{T(i), T(i')\}$ et $\{T(j), T(j')\}$,
- en ajoutant les arêtes $\{T(i), T(j)\}$ et $\{T(i'), T(j')\}$.

Question 8.7. Faire un dessin.

Question 8.8. Écrire la fonction `deux_opt`.

Question 8.9. Écrire la fonction `deux_opt_iterative` qui construit un tour en choisissant un tour initial puis en l'appliquant la transformation *2-opt* tant qu'il existe une transformation *2-opt* qui transforme le tour en un tour de poids strictement plus petit. (Faites bien attention à la complexité!)

8.5 Évaluation et séparation

Dans cette partie, on résout de façon exacte le problème du voyageur de commerce par une recherche (presque) exhaustive parmi tous les tours.

Pour énumérer tous les tours, on écrit une fonction récursive qui prend en paramètre un début de tour composé de p premiers sommets et qui renvoie le meilleur tour qui commence par ces p sommets. Pour cela, elle tente d'ajouter un $p + 1$ -ème sommet parmi ceux encore disponibles et se rappelle récursivement pour compléter le tour. Elle effectue la tentative d'ajout pour tous les sommets possibles et ne conserve que celui de poids minimum.

Question 8.10. Écrire la fonction `tour_backtrack`. Les deux paramètres `meilleur_tour` et `poids_meilleur` sont des pointeurs vers des zones mémoires que la fonction modifiera pour stocker le meilleur tour qu'elle a trouvé ainsi que son poids.

On suppose que `tour_partiel` est déjà allouée et permet de stocker tous les sommets du tableau.

En cours de construction des tours, l'algorithme dispose déjà d'un candidat meilleur tour. Il est possible que le tour partiel soit de poids déjà supérieur au candidat meilleur tour, ou bien qu'on soit en mesure d'estimer qu'avec les arêtes qu'il reste à ajouter, le tour partiel finira par dépasser le candidat meilleur tour. Dans ce cas, il est inutile de continuer la construction d'un tour à partir de ce tour partiel.

Si C est un tour partiel, on construit une fonction *eval* telle que, pour tout tour T qui contient C , $poids(T) \geq eval(C)$. Cette fonction fournira un bon estimateur pour arrêter les recherches à partir de C .

Soit c_0 le premier sommet de C . On note $eval_0(C)$ le poids de la plus petite arête qui relie c_0 à un sommet qui n'est pas dans C .

Soit c_p le dernier sommet de C . On note $eval_1(C)$ le poids de la plus petite arête qui relie c_p à un sommet qui n'est pas dans C .

Soit $v \notin C$. On considère les deux arêtes de poids minimum dont une extrémité est v et l'autre est soit le premier sommet de C , soit le dernier sommet de C , soit un sommet qui n'est pas dans C . On note $eval_2(C, v)$ la somme des poids de ces arêtes.

Enfin, on définit :

$$eval(C) = poids(G) + \left\lceil \frac{1}{2} \left(eval_0(C) + eval_1(C) + \sum_{v \notin C} eval_2(C, v) \right) \right\rceil$$

Question 8.11. Montrer la relation $poids(T) \geq eval(C)$ si T est un tour qui commence par les mêmes sommets que C .

Question 8.12. Écrire la fonction `somme_deux_plus_legeres` qui, pour un sommet v donné et un tour partiel C calcule $eval_2(C, v)$.

Question 8.13. Écrire la fonction `eval_tour_partiel` qui calcule $eval(C)$.

Question 8.14. À partir du code de `tour_backtrack`, écrire `tour_minimum` qui améliore le temps de recherche avec la fonction d'évaluation.

TP 9

Automates finis

On considère des mots sur un alphabet Σ . On pourra représenter dans ce sujet les lettres par le type `char` et les mots par des chaînes de caractères de type `char *`, terminées par le caractère nul. Un automate est un modèle de machine qui permet de reconnaître linéairement en la taille d'un mot s'il correspond à un motif fixé (spoiler : une expression régulière).

9.1 Représentation d'une machine à états

Un automate est une machine qui à chaque instant se trouve dans un *état* donné. On note Q l'ensemble de tous les états. Dans ce sujet, on considère qu'il s'agit des entiers dans $\llbracket 0, |Q| - 1 \rrbracket$.

On représente en C un automate fini par une valeur du type `automaton`, donnée dans le fichier `automata.h`.

Le champ `first_transition` est un tableau qui pour chaque lettre a indique la position, dans `transitions` de la première transition étiquetée par a . Ainsi, si a est une lettre, les transitions étiquetées par a se trouvent entre les cases d'indices `first_transition[a]` et `first_transition[a+1] - 1`. La toute dernière case du tableau `first_transition` permet de conserver cette règle y compris pour la dernière lettre de l'alphabet.

Question 9.1. Que se passe-t-il quand aucune transition n'est étiquetée par une lettre ?

Certains états sont dits *acceptants*. Cette information est indiquée dans le champ `is_final`. Initialement, quand il n'a encore lu aucune lettre, l'automate est dans l'état 0.

9.2 Automate déterministe

On suppose dans cette partie que les automates sont déterministes.

Question 9.2. Écrire une fonction `automaton_accepts` qui renvoie un booléen indiquant si, en partant de l'état 0 et en lisant un mot, un automate termine son calcul dans un état acceptant. Si l'automate est bloqué en cours de calcul (car il n'existe aucune transition portant la bonne étiquette), la fonction renvoie `false`.

(Indication : écrire d'abord comment, étant donné un état, une lettre et un automate calculer la valeur de l'état suivant s'il existe.)

Question 9.3 (Le totomate). Décrire un automate déterministe tel que les mots acceptés sont exactement ceux qui contiennent le facteur `toto`. On pourra remarquer qu'un automate se dessine naturellement comme un graphe étiqueté dont les nuds sont les états et les arêtes sont les transitions.

Question 9.4. Décrire un automate qui accepte exactement les mots de $L_0 = \{a^n b^m \mid n, m \in \mathbb{N}\}$.

Question 9.5. Décrire un automate qui accepte exactement les mots de L_0^* .

Question 9.6. Qui n'a pas programmé en C les tests avec ces automates ?

9.3 Lecture d'un automate

Au lieu d'écrire un automate directement en C, j'aimerais écrire mes automates sous la forme d'un fichier texte :

```
finals: 4;
0 ->[t] 1;
1 ->[o] 2;
1 ->[t] 1;
2 ->[t] 3;
3 ->[t] 1;
3 ->[o] 4;
4 -> 4;
0 ->![t] 0;
1 ->![to] 0;
2 ->![t] 0;
3 ->![to] 0;
```

La première ligne indique la liste des états acceptants, séparés par des espaces. Chaque ligne suivante est en général de la forme `q1 ->[lettre] q2;` pour une transition $q_1 \rightarrow^a q_2$. La lettre peut ne pas être indiquée, on a alors une ligne `q1 -> q2;` qui indique que l'on crée *toutes* les transitions $q_1 \rightarrow^a q_2$, lorsque a décrit l'alphabet. Enfin, on peut également écrire des lignes de la forme `q1 ->[lettres] q2;!` pour indiquer que l'on crée les transitions de q_1 vers q_2 pour toutes les lettres sauf celles indiquées dans les crochets.

Le programme principal s'utiliserait alors ainsi :

```
./automaton fichier_automate
```

Ce programme lit sur l'entrée standard des mots, un par ligne, et pour chacun d'entre eux indique s'il est reconnu par l'automate.

Question 9.7. Écrire la fonction qui, à partir d'un tel fichier, crée la structure de type `automaton` correspondante. (C'est une fonction un brin technique.)

9.4 Langages locaux

Question 9.8. Soit L un langage local. Construire un automate qui reconnaît exactement les mots de L . Votre automate devrait avoir $|\Sigma| + 1$ états et vérifier le fait que toutes les transitions étiquetées par une même lettre partent d'un même état.

Question 9.9. Réciproquement, montrer qu'un automate tel que toutes les transitions étiquetées par une même lettre partent d'un même état reconnaît exactement les mots d'un langage local, à définir.

9.5 Automates non déterministes

Désormais, on considère des automates qui peuvent être non déterministes. L'automate accepte alors un mot *s'il existe au moins un chemin* étiqueté par les lettres du mot, allant de l'état initial jusqu'à un état final. S'il existe d'autres chemins (non acceptants ou bloqués), ces autres chemins ne sont pas pris en compte.

Question 9.10. Écrire une fonction `automaton_nd_accepts` qui explore tous les chemins et indique si le mot est accepté. Une fonction intermédiaire utile pourrait être la fonction `automaton_nd_accepts_from` qui indique si un mot est accepté mais en partant d'un état quelconque fixé, pas nécessairement 0.

Une deuxième approche pour explorer tous les chemins consiste à garder en mémoire, à chaque instant, la liste des états dans lesquels l'automate se trouve. Ceci revient à faire avancer en parallèle tous les chemins en cours d'exploration. Une telle liste peut se représenter par un tableau qui à chaque état associe un booléen.

Question 9.11. Écrire une fonction `automaton_nd_accepts_parallel` qui utilise cette idée.

TP 10

Algorithme de McNaughton et Yamada

Dans ce sujet, on part d'un automate \mathcal{A} et on construit une expression régulière qui reconnaît $L(\mathcal{A})$ avec l'algorithme de McNaughton et Yamada.

10.1 Description de l'algorithme

On note $L_{p,q}$ l'ensemble de tous les mots w tels que $\delta^*(p, w) = q$.

Question 10.1. Montrer que $L_{p,q}$ est reconnaissable par automate. Exprimer $L(\mathcal{A})$ en fonction des $L_{p,q}$.

On note n le nombre d'états de l'automate et on identifie les états avec les entiers de $\llbracket 0, n-1 \rrbracket$. On note $L_{p,q}^k$ l'ensemble de tous les mots w tels qu'il existe un chemin dans l'automate de l'état p à l'état q , étiqueté par w et dont tous les états intermédiaires (donc tous les états rencontrés, sauf les extrémités p et q) sont strictement inférieurs à k .

Question 10.2. Expliciter $L_{p,q}^0$. Reconnaître $L_{p,q}^n$ (attention au cas $p = q$).

Question 10.3. Déterminer une formule de récurrence qui exprime $L_{p,q}^k$ en fonction de certains $L_{i,j}^{k-1}$.

(Indication : elle devrait vous faire penser à un algorithme de MP2I, vu dans un contexte qui n'avait à priori rien à voir avec les automates.)

Question 10.4. En déduire un algorithme qui calcule une expression régulière décrivant $L(\mathcal{A})$.

Question 10.5. Appliquer cet algorithme sur des petits automates.

L'expression régulière obtenue dépend fortement de la numérotation des sommets choisie, et n'est pas toujours très simple.

Question 10.6. Quel résultat théorique est démontré par cet algorithme ? (Indice : c'est une inclusion entre certains ensembles.)

10.2 Simplification d'expressions régulières

On représente en OCaml les expressions régulières par des valeurs de type :

```
type regexp =  
  | Vide  
  | Epsilon  
  | Mot of string  
  | Union of regexp * regexp  
  | Concat of regexp * regexp  
  | Etoile of regexp
```

Question 10.7. Écrire une fonction `est_vide` qui décide si une expression régulière engendre le langage vide.

On dit que deux expressions régulières sont *équivalentes* lorsqu'elles engendrent le même langage. On note \equiv cette relation.

On a notamment les équivalences suivantes, quelle que soit l'expression E :

$$\emptyset \mid E \equiv E \mid \emptyset \equiv E$$

$$\varepsilon \cdot E \equiv E \cdot \varepsilon \equiv E$$

$$\emptyset \cdot E \equiv E \cdot \emptyset \equiv \emptyset$$

$$a \cdot b \equiv ab$$

$$\emptyset^* \equiv \varepsilon$$

$$\varepsilon^* \equiv \varepsilon$$

$$(E^*)^* \equiv E^*$$

Question 10.8. Écrire les fonctions `simplifie_racine_union`, `simplifie_racine_concat` et `simplifie_racine_etoil` qui effectuent les simplifications précédentes, *uniquement* à la racine de l'expression régulière.

Question 10.9. Écrire une fonction `simplifie` qui simplifie le plus possible une expression régulière, en utilisant les équivalences précédentes.

10.3 Programmation de l'algorithme de McNaughton et Yamada

On représente en OCaml un automate par le type suivant :

```
type état = int
type transition = {
    debut : état;
    etiq : char;
    fin : état;
}
type automate = {
    nb_etats: int;
    initiaux: état list;
    finaux: état list;
    transitions: transition list;
}
```

Question 10.10. Écrire la fonction `mny` qui prend en paramètre un automate et renvoie une expression régulière engendrant le langage de l'automate, en utilisant l'algorithme de McNaughton et Yamada.

10.4 Pour aller plus loin

Il existe d'autres algorithmes qui résolvent le même problème. Les deux autres algorithmes habituels sont l'algorithme de Conway et l'algorithme de Brzozowski et McCluskey.

L'algorithme de Conway est très bien décrit dans la partie 2 du sujet CentraleSupélec 2022 (option informatique).

10.5 Algorithme de Brzozowski et McCluskey

On généralise la notion d'automate non déterministe en étiquetant désormais les transitions non plus uniquement par des lettres mais par des expressions régulières. Si $q \rightarrow^e q'$ est une transition de l'automate étiquetée par une expression régulière e , alors la relation de transition donne $\delta(q, u) = q'$ dès lors que $u \in L(e)$.

Quand les expressions régulières sont uniquement des lettres ou ε , on retrouve les automates non déterministes avec transitions spontanées.

Question 10.11. Montrer que tout langage régulier est reconnu par un automate généralisé possédant exactement deux états.

Question 10.12. On considère un automate généralisé possédant deux transitions $q \rightarrow^{e_1} q'$ et $q \rightarrow^{e_2} q'$. Construire un automate qui reconnaît le même langage et qui possède une transition de moins que l'automate d'origine. Cette étape, que l'on réalise tant que l'hypothèse est vraie, est appelée *élimination des transitions*.

Question 10.13. On considère un automate généralisé dans lequel on a effectué l'élimination des transitions. On suppose qu'il possède deux transitions $q \rightarrow^{e_1} q'$ et $q' \rightarrow^{e_2} q''$, et peut-être une transition $q' \rightarrow^e q'$. Construire un automate qui reconnaît le même langage et qui possède strictement moins de transitions que l'automate d'origine.

Question 10.14. Dans la transformation précédente, on fixe q et on applique tant que c'est possible la transformation précédente. Montrer qu'il existe un automate possédant un état de moins et qui reconnaît le même langage. Cette étape est appelée *élimination d'un état*.

Question 10.15. À partir d'un automate, donner un algorithme qui construit un automate généralisé reconnaissant le même langage et possédant exactement deux états et une transition.

TP 11

Chemin hamiltonien probabiliste

11.1 De l'aléatoire

La bibliothèque C permet de générer des entiers aléatoires pris dans l'ensemble $\llbracket 0, \text{RAND_MAX} \rrbracket$. On suppose que ce tirage est uniforme.

Le générateur aléatoire doit être initialisé avec une graine avant d'être utilisé pour la première fois. La séquence de nombres aléatoires est fixée par la graine, si on initialise à nouveau le générateur avec la même graine, on obtiendra à nouveau la même séquence de nombres.

```
#include <stdlib.h>

/* Obtention d'un entier aléatoire */
int rand(void);
/* Initialisation du générateur */
void srand(unsigned int seed);
```

Une pratique fréquente consiste à utiliser l'heure actuelle pour initialiser le générateur :

```
#include <time.h>

void init_alea() {
    srand(time(NULL));
}
```

Ce générateur convient très bien pour des algorithmes probabilistes, mais il n'est pas du tout assez robuste par exemple pour réaliser de la cryptographie ou générer des données nécessitant de vraies propriétés d'aléatoire.

Question 11.1. On veut simuler une loi uniforme sur un ensemble $\llbracket 0, N \rrbracket$. Pourquoi le code suivant est-il incorrect ?

```
int uniforme(int n) {
    return rand() % n;
}
```

11.2 Générer un graphe aléatoire, modèle d'Erds et Rényi

Soit p un réel strictement entre 0 et 1. On représente les graphes par des listes d'adjacence. On note $G_{n,p}$ un graphe à n sommets tel que chaque arête est présente avec probabilité p .

Question 11.2. Écrire une fonction `graphe_genere` qui génère un tel graphe.

Question 11.3. Donner l'espérance du nombre d'arêtes de $G_{n,p}$.

On aura besoin de considérer que les arêtes d'un sommet sont réparties uniformément au hasard dans chaque liste d'adjacence. Pour cela, on va mélanger les listes d'adjacence.

Question 11.4. Écrire une fonction `mélange_liste` qui mélange en place une liste d'adjacence. (On pourra utiliser par exemple l'algorithme de Knuth, Fisher et Yates.)

Question 11.5. Modifier la fonction `graphe_genere` afin qu'elle renvoie des listes d'adjacence mélangées.

Erds et Rényi proposent un deuxième modèle de graphes aléatoires, noté $G_{n,m}$ où n est un nombre de sommets fixé et m est le nombre d'arêtes à créer. Un tel graphe est obtenu en tirant exactement m arêtes parmi les $\binom{n}{2}$ arêtes possibles. Ceci peut se faire en tirant les arêtes les unes après les autres.

Question 11.6. On considère $\mathcal{E} = \{(a, b) \in \llbracket 0, n-1 \rrbracket^2 \mid a > b\}$ et φ la fonction définie sur \mathcal{E} par $\varphi(a, b) = \binom{b}{1} + \binom{a}{2}$. Montrer qu'elle réalise une bijection (quel est l'ensemble image?).

Question 11.7. Écrire une fonction `graphe_genere_aretes` qui génère un tel graphe. On prendra soin de bien choisir uniformément les arêtes.

Dans la suite, on travaillera avec un graphe $G_{n,p}$.

11.3 Un premier algorithme de chemin hamiltonien

On considère un chemin élémentaire du graphe, formé des sommets v_1, \dots, v_k dans cet ordre. Sur un tel chemin, on va réaliser deux opérations :

- l'ajout d'un sommet en fin de chemin,
- la *rotation* $\text{ROTATION}(v_i, v_k)$. C'est l'opération qui transforme ce chemin en $v_1, \dots, v_i, v_k, v_{k-1}, \dots, v_{i+1}$, c'est-à-dire qu'elle retire l'arête entre v_i et v_{i+1} et ajoute au chemin l'arête entre v_i et v_k .

Question 11.8. Proposer une structure pour représenter un chemin qui permet de réaliser ces opérations. Implémenter la fonction `rotation` correspondante.

(Un $O(n)$ est facile à voir. Je pense qu'on peut arriver en $O(1)$.)

Question 11.9. Implémenter la fonction qui permet d'ajouter un sommet à la fin du chemin.

À chaque étape de l'algorithme, on garde en mémoire pour chaque sommet une liste d'arêtes adjacentes à ce sommet et qui n'ont pas encore été utilisées. On note, pour chaque sommet v , `unused(v)` cette liste. Initialement, elle correspond à la liste d'adjacence de v .

Pour tout $v \in V$ **Faire**

`unused(v) ← {(v, u) | (v, u) ∈ E}`

Fin pour

Prendre un sommet au hasard, en faire la tête du chemin

Tant que on n'a pas trouvé de solution, **faire** :

Soit (v_1, \dots, v_k) le chemin actuel

Si `unused(vk) = ∅` **alors**

Échouer

Fin si

Soit (v_k, u) la première arête de `unused(vk)`

Retirer (v_k, u) de `unused(vk)`

Retirer (u, v_k) de `unused(u)`

Si $u \notin \{v_1, \dots, v_{k-1}\}$ **alors**

Ajouter u à la fin du chemin

Sinon

Soit i tel que $u = v_i$

Faire une rotation entre v_k et v_i . Le dernier sommet du chemin devient v_{i+1} .

Fin si

Fin tant que

Question 11.10. Implémenter cet algorithme. Comment sait-on si on a terminé?

11.4 Intermède mathématique

Si X est une variable aléatoire, si $a \geq 0$ et si $t > 0$, alors l'inégalité de Chernoff dit :

$$P(X \geq a) \leq \frac{E(e^{tX})}{e^{ta}}$$

$$P(X \leq a) \leq \frac{E(e^{-tX})}{e^{-ta}}$$

Cela sert souvent en informatique, ainsi que les inégalités de Markov et de Bienaymé-Tchebychev.

11.5 Preuve de validité

L'algorithme précédent n'est pas facile à analyser car les choix effectués à une étape conditionnent les choix suivants. On introduit un algorithme modifié, qui effectue des étapes qui ne servent pas à améliorer le chemin trouvé, mais qui garantissent que les étapes de l'algorithme deviennent indépendantes.

Prendre un sommet au hasard, en faire la tête du chemin

Tant que on n'a pas trouvé de solution, **faire** :

 Soit (v_1, \dots, v_k) le chemin actuel

Si avec probabilité $1/n$ **alors**

 Retourner le chemin

Sinon, si avec probabilité $\text{card}(\text{used}(v_k))$

 Choisir une arête (v_k, v_i) au hasard dans $\text{used}(v_k)$

 Faire la rotation (v_k, v_i) , la fin du chemin devient v_{i+1}

Sinon

 Soit (v_k, u) la première arête de $\text{unused}(v_k)$

 Retirer (v_k, u) de $\text{unused}(v_k)$

 Retirer (u, v_k) de $\text{unused}(u)$

Si $u \notin \{v_1, \dots, v_{k-1}\}$ **alors**

 Ajouter u à la fin du chemin

Sinon

 Soit i tel que $u = v_i$

 Faire une rotation entre v_k et v_i . Le dernier sommet du chemin devient v_{i+1} .

Fin si

Fin si

Fin tant que

On veut montrer que sur un graphe $G_{n,p}$, si $p \geq \frac{40 \ln(n)}{n}$ alors $P(\text{trouver un chemin hamiltonien}) \geq 1 - O(\frac{1}{n})$.

On va commencer par étudier une variante des graphes aléatoires, et on verra plus tard comment on revient à $G_{n,p}$. Dans un premier temps, on fixe une probabilité q . Pour construire $\text{used}(v)$, on considère les $n - 1$ arêtes possibles, et on les insère dans $\text{used}(v)$ avec probabilité $1 - q$. L'ensemble d'arêtes complémentaire est noté $\text{unused}(v)$. Ceci implique en particulier qu'une arête (u, v) peut être placée dans $\text{unused}(u)$ mais pas dans $\text{unused}(v)$, bien que le graphe soit non orienté.

Question 11.11. Montrer que tant que l'algorithme modifié n'a pas échoué, à chaque étape, tous les sommets ont la même probabilité de devenir le dernier sommet du chemin partiel, peu importe les choix précédents.

On suppose que $q \geq \frac{20 \ln(n)}{n}$.

Soit E_1 l'événement qui est réalisé lorsque l'algorithme a effectué $3n \ln(n)$ tours de boucle sans échouer et sans trouver de chemin hamiltonien.

Soit F_u l'événement qui est réalisé lorsque le sommet u n'a pas été visité lors des $2n \ln(n)$ premières étapes de l'algorithme.

Question 11.12. Montrer que $P(F_u) \leq (1 - \frac{1}{n})^{2n \ln(n)} \leq \frac{1}{n^2}$. En déduire que la probabilité de ne pas avoir visité tous les sommets en $2n \ln(n)$ étapes est majorée par $1/n$.

Question 11.13. En déduire que $P(E_1) = O(\frac{1}{n})$.

On définit les évènements :

- E_2 : l'algorithme a échoué en moins de $3n \ln(n)$ étapes,
- E_{2a} : il existe un k tel que $\text{unused}(v_k)$ a perdu au moins $9 \ln(n)$ arêtes au cours des $3n \ln(n)$ premières étapes,
- E_{2b} : il y avait au départ un v_k avec $\text{card}(\text{unused}(v_k)) \leq 10 \ln(n)$.

Question 11.14. Remarquer que $P(E_2) \leq P(E_{2a}) + P(E_{2b})$.

On va majorer chaque terme par $1/n$.

Question 11.15. Soit X_k la variable aléatoire qui donne le nombre de fois où le sommet v_k s'est retrouvé au bout du chemin partiel. Reconnaître la loi de X_k . Montrer que $P(X_k \geq 9 \ln(n)) \leq \frac{1}{n^2}$. En déduire que $P(E_{2a}) \leq \frac{1}{n}$.

(Indication. Chernoff en $\ln(n)/3$.)

Question 11.16. Soit Y_k le nombre d'arêtes d'une liste $\text{unused}(v_k)$ initialement. Reconnaître la loi de Y_k . En utilisant un raisonnement similaire qu'à la question précédente, montrer que $P(E_{2b}) \leq \frac{1}{n}$.

(Indication. $t = -\ln(n)$.)

On revient désormais à un graphe $G_{n,p}$. On choisit q tel que $p = 2q - q^2$. À partir d'un graphe $G_{n,p}$, on initialise les listes used et unused de la façon suivante :

- l'arête (u, v) est placée dans $\text{unused}(u)$ uniquement avec probabilité $p \frac{q(1-q)}{2q-q^2}$,
- l'arête (u, v) est placée dans $\text{unused}(v)$ uniquement avec probabilité $p \frac{q(1-q)}{2q-q^2}$,
- l'arête (u, v) est placée dans les deux avec probabilité $p \frac{q^2}{2q-q^2}$.

Question 11.17. Vérifier qu'une arête est placée dans $\text{unused}(u)$ avec probabilité q . Vérifier que c'est indépendant du fait qu'elle soit placée dans $\text{unused}(v)$.

Et donc le résultat précédent s'applique.

Question 11.18. Conclure.

TP 12

Assistant de preuve

12.1 Renommer les variables liées

OCaml dispose d'un module `Map` qui permet de construire des tables, immuables, qui à des clés associent des valeurs. Les clés doivent être prises dans un ensemble ordonné, c'est-à-dire que l'on doit disposer d'un module qui a la signature suivante :

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end
```

Le module `Int` (et ce n'est pas le seul) possède ces éléments. Ainsi, il est possible d'écrire :

```
module IntMap = Map.Make(Int)
```

On dispose alors d'un nouveau module, nommé `IntMap` qui définit un type `'a IntMap.t`. Les objets de ce type sont tables qui à des entiers associent des valeurs de type `'a`. Voici quelques fonctions du module `IntMap` :

```
val empty : 'a IntMap.t
val is_empty : 'a IntMap.t -> bool
val mem : int -> 'a IntMap.t -> bool
val add : int -> 'a -> 'a IntMap.t -> 'a IntMap.t
val remove : int -> 'a IntMap.t -> 'a IntMap.t
val find : int -> 'a IntMap.t -> 'a
```

L'ensemble des fonctions d'un module fabriqué par `Map.Make` est documenté dans le manuel : <https://v2.ocaml.org/api/Map.S.html>.

Question 12.1. On dit que deux formules sont α -équivalentes lorsqu'elles diffèrent uniquement sur le choix des noms des variables liées. Ainsi, $\forall x, x \rightarrow y$ et $\forall t, t \rightarrow y$ sont α -équivalentes. Elles ne sont pas α -équivalentes avec $\forall x, x \rightarrow z$.

Écrire la fonction `Prooftree.alpha_equiv` qui teste cette équivalence.

Pour une formule donnée, on dit qu'une variable est *fraîche* s'il s'agit d'une variable qui n'apparaît pas du tout dans la formule.

On définit une *substitution* σ comme étant une fonction qui à certaines variables x_0, \dots, x_{n-1} associe respectivement des formules F_0, \dots, F_{n-1} . Si F est une formule, l'application de σ à F , notée $F[\sigma]$ consiste à remplacer chaque occurrence d'une variable libre x_i par F_i .

Il faut faire attention au phénomène de *capture* : chaque formule F_i peut contenir ses propres variables libres, il faut éviter que, lors de la substitution, ces variables ne soient liées par un quantificateur dans F .

Ainsi, si $F = \forall x, y$ et si $\sigma(y) = x$, alors $F[\sigma]$ n'est pas $\forall x, x$. À la place, on peut d'abord renommer x en une variable fraîche z , alors $F = \forall z, y$, puis $F[\sigma] = \forall z, x$.

Question 12.2. On note $FV(F)$ les variables libres d'une formule. Discuter, en fonction des valeurs de $FV(F)$ et $FV(G)$, comment réaliser une substitution $(\forall y, F)[x \mapsto G]$.

Question 12.3. Programmer la fonction `Prooftree.substitute`.

12.2 Vérificateur d'arbre de preuve

Dans cette partie, on suppose que l'on dispose d'un candidat arbre de preuve et on veut écrire un programme qui décide si cet arbre est bien formé.

Question 12.4. Les déclarations sont dans `prooftree.mli`, il ne vous reste qu'à écrire la fin de `prooftree.ml`. Les types doivent être déclarés dans le fichier `prooftree.mli` et dans le fichier `prooftree.ml`. N'oubliez pas de recopier leur définition.

Question 12.5. Pour la réalisation de tests, on se partage le travail. Chacun dans la salle écrit un arbre de preuve valide et un arbre de preuve invalide. Je regroupe l'ensemble dans un fichier de tests.

12.3 Résoudre l'unification

Dans cette partie, on s'intéresse uniquement aux formules sans aucun quantificateur.

Le problème d'unification consiste à décider, pour deux formules F et G , s'il existe une substitution σ telle que $F[\sigma] = G[\sigma]$. C'est un problème fondamental pour un grand nombre d'autres questions concernant la vérification de types, la vérification de preuves ou la recherche automatique de preuves. Si avez un peu de curiosité, vous pourrez regarder par exemple chez vous une introduction au langage Prolog, qui fait un usage important de l'unification.

Le but est de programmer un algorithme qui à partir d'un ensemble de *contraintes* trouve une substitution σ qui satisfait les contraintes. Une contrainte est une égalité de la forme $x = y$ où x et y sont des variables, ou bien de la forme $x = F$ où F est une formule logique (plus compliquée qu'une seule variable). À gauche de l'égalité, on ne trouve toujours qu'une variable seule. Dans un ensemble de contraintes valide, on interdit l'existence de cycles sur une même variable, obtenus par transitivité de l'égalité. On interdit également la présence de deux définitions pour une même variable.

Une première étape consiste, à partir de deux formules logiques F et G , à générer une liste de contraintes nécessaires pour avoir l'égalité $F = G$. Par exemple, si $F = (x \rightarrow y) \rightarrow z$ et $G = w \rightarrow w \rightarrow w$, alors l'égalité $F = G$ génère les contraintes suivantes :

$$w = x \rightarrow y \quad z = w \rightarrow w$$

De manière générale, soient C_1 et C_2 deux connecteurs logiques. Une égalité $C_1(F_1, \dots, F_n) = C_2(G_1, \dots, G_n)$ ne peut être satisfaite que si $C_1 = C_2$ et si, récursivement, les égalités $F_i = G_i$ peuvent être satisfaites.

Question 12.6. À quoi peut servir *union-find* pour résoudre les contraintes ? Le module, fourni, `Constraints` semble donc bien pratique pour traiter la suite.

Question 12.7. Écrire une fonction `constraints` qui à partir de deux formules génère la liste des contraintes. En déduire la fonction `add_constraints` qui ajoute ces contraintes à l'ensemble des contraintes déjà existantes. Signalez les cas d'unification impossible avec des exceptions OCaml (que vous définirez).

Question 12.8. Écrire la fonction `resolve` puis la fonction `unify`.

TP 13

Analyse d'un mini langage

13.1 Mini langage des expressions arithmétiques

Dans ce TP, on écrit un analyseur syntaxique pour un langage d'expressions arithmétiques composé des opérateurs `*`, `+`, `-`, `/`, de parenthèses, de variables et d'entiers.

13.2 Analyse lexicale

L'ensemble des structures à manipuler sont définies dans le fichier `minilexer.h`. Vous aurez également besoin de compléter les fichiers `minilangage.h` et `minilangage.c`. Le but est de transformer une suite de caractères (dans un fichier donnée en entrée) en une suite de lexèmes. Ces éléments seront les lettres données ensuite à la grammaire pour l'analyse syntaxique.

On va utiliser les expressions POSIX étendues de la bibliothèque standard C. On les trouve dans `regex.h`. Une expression régulière est représentée par une valeur de type `regex_t`. On crée une telle expression avec la fonction `regcomp` :

```
regex_t regex;
regcomp(&regex, "[0-9]+", REG_EXTENDED | REG_NEWLINE);
```

Question 13.1. Dans les fichiers `minilangage.*`, définir les expressions régulières utiles pour reconnaître les éléments du mini langage des expressions arithmétiques. Écrire les fonctions de création des lexèmes correspondantes.

On pourra utiliser la fonction `strtol` (dans `stdlib.h`), qui permet de convertir une chaîne de caractères `nptr` encodant un entier dans une base `base` donnée en un entier :

```
strtol(nptr, NULL, base);
```

Question 13.2. L'analyseur lexical doit ignorer les espaces, les retours à la ligne. Écrire la regex `whitespace` correspondante.

Une fois une regex créée, on peut vérifier si une chaîne correspond à la regex avec l'appel :

```
regex_t regex;
regmatch_t match[1];
regexexec(&regex, chaine, 1, match);
```

La fonction renvoie 0 si la chaîne est reconnue par l'expression et une valeur `REG_NOMATCH` sinon. La variable `match[0]` sert à stocker les informations sur la reconnaissance de la chaîne. Elle est de type `regmatch_t` : c'est une structure qui possède deux champs :

- `rm_so` indique l'indice du premier caractère reconnu,
- `rm_eo` indique l'indice du dernier caractère reconnu.

Question 13.3. Écrire la fonction `lexer_next` dans `minilexer.c` qui prend en paramètre un lexer et trouve le lexème suivant. Elle doit ignorer les espaces grâce à la regex `whitespace`. (Indication : pour lire des données depuis un fichier, on pourra utiliser `fgets`.)

Question 13.4. Faites des tests, c'est indispensable. Créez quelques fichiers avec des expressions arithmétiques. Affichez le résultat du lexer.

13.3 Analyse syntaxique

On considère d'abord la grammaire suivante :

$$S \rightarrow \text{num} \mid \text{num} + S$$

Un mot engendré par cette grammaire est naturellement une liste d'entiers.

Question 13.5. Écrire une fonction `minilangage_parser_plus` qui reconnaît un mot de cette grammaire et qui renvoie la liste d'entiers. Elle signale les erreurs de syntaxe si elle en rencontre.

On considère désormais le langage complet décrit en introduction.

Question 13.6. Définir en C un type `expression` qui permet de représenter une expression de ce langage.

On considère la grammaire simplifiée :

$$S \rightarrow \text{operand} + \text{operand} \mid \text{operand} - \text{operand} \mid \text{operand} * \text{operand} \mid \text{operand} / \text{operand}$$

$$\text{operand} \rightarrow \text{id} \mid \text{num} \mid (S)$$

Pour reconnaître des mots de cette grammaire, on conseille d'avoir deux fonctions auxiliaires récursives `minilangage_parse_operand` et `minilangage_parse_expression` qui vont prendre en paramètre un lexer.

Question 13.7. Écrire une fonction `minilangage_parser_noprio` qui reconnaît un mot de cette grammaire et qui renvoie l'expression reconnue, en signalant si besoin les erreurs de syntaxe.

Question 13.8. Définir à présent la grammaire complète, en prenant en compte les priorités des opérateurs. Écrire la fonction `minilangage_parser` qui construit l'expression.

13.4 Commentaires

Question 13.9. Ajouter dans l'analyseur syntaxique la gestion de commentaires (par exemple au format C `/* */`).

13.5 Liaison de variables

Question 13.10. Ajouter le support pour une expression de la forme `let x = expr in expr` qui permet de lier une variable.

Question 13.11. Écrire la fonction qui prend en paramètre l'arbre de syntaxe abstraite d'une expression et qui vérifie que toutes les variables utilisées se trouvent bien dans le corps d'un `let` qui les lie à une valeur.

13.6 Pression de registres

Pour chaque sous expression, il existe un certain nombre de variables liées utilisables dans cette sous expression. Ce nombre n'est pas le même pour toutes les sous expressions. Considérez par exemple :

```
let x = 32 in (let y = 52 in x * y) * x * (let z = 42 in z)
```

Dans la sous expression $x * y$, il y a deux variables liées tandis que dans la sous expression commençant au `in` du `let x`, il n'y a qu'une seule variable liée.

Un processeur dispose de *registres*, qui sont des éléments de mémoire qui permettent de stocker un entier (donc la valeur d'une variable de notre langage). Les registres sont en quantité limitée, donc on veut en utiliser le moins possible. Il faut décider pour chaque variable dans quel registre elle sera stockée.

Dans l'exemple précédent, il existe une solution utilisant uniquement deux registres, où x utilise un registre et où y et z , qui n'existent pas simultanément, peuvent être affectées au même registre.

Question 13.12. Déterminer, à partir d'un arbre de syntaxe abstraite, le plus petit nombre de registres nécessaires pour stocker à chaque instant toutes les variables. (Indication : c'est un problème bien connu dans un autre chapitre.)

TP 14

Être à la mode

14.1 Code déjà tout prêt

Une archive contenant du code pré-écrit en OCaml et les fichiers de données est disponible dans tous les points de vente habituels. Vous en avez besoin pour traiter ce sujet.

14.2 Base du MNIST

La base de données du MNIST est un jeu de données contenant des chiffres manuscrits, que tout le monde utilise en apprentissage comme jeu de données usuel. Cette base fournit en fait deux jeux de données :

- un jeu de 60000 images, avec les chiffres associés, destinées à l'entraînement,
- un jeu de 10000 images, avec les chiffres associés, destinées aux tests.

Les images sont des carrés de 28 pixels de côté.

Ces données sont accessibles à l'adresse <http://yann.lecun.com/exdb/mnist/>, où l'on trouvera également une description précise du format des fichiers. Le format est très simple à lire : je vous invite très fortement à implémenter vous-mêmes à titre d'exercice des fonctions de lecture des fichiers IDX en C et en OCaml (mais ce n'est pas l'objet du TP aujourd'hui).

La base de données Fashion MNIST (<https://github.com/zalandoresearch/fashion-mnist>) suit exactement le même format que la base d'origine du MNIST sauf que les images représentent des vêtements au lieu de représenter des chiffres. C'est avec cette base de données que l'on travaille dans ce sujet.

Vous avez à votre disposition un module `Mnist`. Lisez son interface dans `mnist.mli`. Ses fonctions permettent de lire des données contenues dans un fichier IDX. La lecture d'une image donne des données dans un tableau d'entiers à une dimension, contenant l'image ligne par ligne.

Question 14.1. Écrire une fonction qui affiche une représentation d'une image à l'écran, à partir de son index dans la base du MNIST. Dans les données, chaque pixel est un entier compris entre 0 (blanc) et 255 (noir). On peut choisir un jeu de caractères qui représentera chaque luminosité : le fichier `grayscale_70_levels.txt` vous suggère une liste de 70 caractères classiques pour représenter les tons allant du blanc au noir en ASCII art. Vous pouvez utiliser n'importe quelle autre représentation.

On va tenter de lire les données de manière paresseuse, c'est-à-dire sans nécessairement toujours charger les 60000 symboles en mémoire. Au lieu d'utiliser des listes, on utilise le type OCaml `Seq.t` du module `Seq`. Il est défini de la façon suivante :

```
type 'a t = unit -> 'a node
type 'a node =
  | Nil
  | Cons of 'a * 'a t
```

Un `Seq.t` est une fonction, qui prend en paramètre un `unit`. L'élément suivant de la séquence n'est donc obtenu qu'en le demandant explicitement par un appel de fonction. Cette appel de fonction renvoie

soit `Nil` pour indiquer que la séquence est terminée, soit `Cons(x,f)` où `x` est l'élément suivant et `f` est une fonction qui, quand on l'appellera, donnera la suite de la séquence.

14.3 File de priorité

Question 14.2. Lisez l'interface du module `PrioQueue` qui se trouve dans le fichier fourni `prioQueue.mli`. Ce module vous donne une implémentation des files de priorité non mutables avec des arbres en tas minimum.

La racine est l'élément minimal pour la fonction de comparaison fournie à la création. Cette fonction respecte la convention usuelle de la fonction `Stdlib.compare` de la bibliothèque standard OCaml. En vous permettant de choisir votre fonction de comparaison, le module vous permet d'utiliser la notion d'ordre qui vous convient pour vos données.

14.4 k plus proches voisins naïf

Le but de cette partie est d'écrire un module `Knn` dans le fichier `knn.ml`, dont l'interface est fixée par `knn.mli`. Vous pouvez vous lancer directement dans l'écriture de ce module sans autre indication, ou bien vous laisser guider par les questions suivantes, qui décomposent le travail.

Question 14.3. Écrire une fonction `count_item` qui compte le nombre d'occurrences d'un élément dans une liste.

Question 14.4. Écrire une fonction `most_frequent` qui détermine, dans une liste non vide, un élément dont le nombre d'occurrences dans la liste est maximal.

Question 14.5. Écrire une fonction `euclidean_dist : float array -> float array -> float` qui calcule la distance euclidienne entre deux vecteurs de \mathbb{R}^d . Les tableaux donnés en entrée sont supposés être de même longueur.

Question 14.6. Écrire une fonction `mnist_list` qui prend en paramètre deux valeurs de type `Mnist.idx`, l'une pour le fichier d'images et l'autre pour le fichier de réponses, et qui construit une liste de couples formées d'une image et de la réponse correspondante (si vous avez compris les séquences, fabriquez une séquence plutôt qu'une liste).

Question 14.7. Écrire une fonction `classify` qui prend en paramètre la liste précédente, un entier `k` et une image du MNIST, qui détermine les `k` plus proches voisins de cette image et renvoie une classe majoritaire parmi ces voisins. On implémentera l'algorithme naïf qui parcourt la liste de toutes les données.

Question 14.8. Tester votre fonction sur les images du jeu de tests : pensez à vous créer un fichier, par exemple `main.ml` qui contient une fonction principale qui lancera tous vos tests. Ce jeu de tests étant fourni avec les réponses, déterminer en fonction de la valeur de k le taux d'erreur de votre reconnaissance. Quelle valeur de k est la plus adaptée sur ces tests ?

14.5 Arbres dimensionnels

Question 14.9. Proposer un type OCaml pour représenter un arbre dimensionnel. (Indication : il n'y a à priori rien de très malin à penser, la réponse est immédiate. Il faut juste penser à stocker les bonnes informations à chaque noeud.)

Question 14.10. Écrire une fonction `dimensional_tree` qui crée l'arbre dimensionnel à partir du jeu de données de Fashion MNIST.

Question 14.11. Améliorer la fonction `init` afin qu'elle construise l'arbre k -dimensionnel puis améliorer la fonction `classify` pour qu'elle utilise cet arbre. Discuter le gain de performance.

Troisième partie

Apartés culturels (HP)

Annexe A

Matroïdes

Sommaire

A.0	Définition et exemples	189
A.1	Algorithme Glouton sur un matroïde	190

Ce chapitre hors programme est présent à titre d'aparté culturel (quoi qu'il ne soit pas impossible que certains énoncés d'oraux, voir d'écrits y fassent référence). Les matroïdes permettent de généraliser un certain nombre de situations déjà rencontrées ou qui seront rencontrées. Ainsi, certains des résultats et certaines des démonstrations ne sont pas bien étonnants et étonnantes en regard des cas particuliers déjà traités (notamment avec l'algorithme de Kruskal qui verra sa correction démontré comme cas particulier).

A.0 Définition et exemples

Définition A.1. Soit S un ensemble d'objets (finie pour les informaticiens). On appelle matroïde un couple (S, I) avec $I \subset \mathcal{P}(S)$ (dont on appelle les éléments des indépendants) qui vérifie les deux propriétés :

(Hérédité) $\forall X \in I, \forall Y \subset X, Y \in I$

(Échange) $\forall (A, B) \in I^2, \#A < \#B \implies \exists x \in B, A \cup \{x\} \in I$

Remarque. Pour tout matroïde (S, I) , on a $\emptyset \in I$. En effet pour $X \in I$ évidemment $\emptyset \subset X$ donc par hérédité $\emptyset \in I$.

Exemple. Soit S l'ensemble des arêtes d'un graphe et I l'ensemble des forêts d'un graphe. S, I est un matroïde :

(Hérédité) si on retire des arêtes à une forêt X , cette dernière reste acyclique et se subdivise en une autre forêt.

(Échange) On distingue selon que $A \subset B$ ou non

Exemple. Soit E un \mathbb{K} -espace vectoriel de dimension finie. Soit S l'ensemble de ses familles libres (qui est non vide par Théorème).

(Hérédité) D'après le cours d'algèbre linéaire, toute sous famille d'une famille libre est elle-même libre.

(Échange) Soit $(A, B) \in I^2$ tel que $\#A < \#B$. La famille $A \cup B$ est une famille génératrice de $F := \text{Vect}(A \cup B) = \text{Vect}(A) + \text{Vect}(B)$. A étant déjà libre, le théorème de la base incomplète assure alors que l'on peut la compléter en une base de F à partir de vecteurs de $A \cup B$ donc de B . D'où l'existence de $x \in B \setminus A$ tel que $A \sqcup \{x\} \in I$.

Proposition A.2. *Tous les indépendants maximaux pour l'inclusion ont le même cardinal.*

Démonstration. Ceci résulte essentiellement de la propriété d'échange. En effet soit $(A, B) \in I^2$. Supposons que $\#A < \#B$, alors $\#(A \cup \{x\}) > \#A$ donc A n'est pas maximal pour l'inclusion et le résultat suit de la contraposition de ce qui vient d'être démontré. ■

A.1 Algorithme Glouton sur un matroïde

Définition A.3. Soit S, I un matroïde. On appelle poids une fonction : $w : S \rightarrow \mathbb{N}$ que l'on étend aux indépendants en posant

$$\forall A \in S, \quad w(A) := \sum_{a \in A} w(a).$$

On a ainsi l'algorithme

Algorithme A.1 Glouton sur un matroïde

Entrée : Un matroïde (S, I)

Sortie : Un indépendant A de poids maximal

fonction GLOUTON $((S, I))$:

$A \leftarrow \emptyset$

$S \leftarrow \text{TRIER}(S, w)$

▷ on trie par poids w décroissant

Pour $s \in S$ dans l'ordre, **faire** :

Si $A \cup \{s\} \in I$ **alors**

$A \leftarrow A \cup \{s\}$

Fin si

Fin pour

renvoyer A

Fin fonction

Lemme A.4. Soit s_k le premier objet sélectionné par l'algorithme glouton. Il existe un indépendant de poids maximal qui contient s_k .

Démonstration. Soit B un indépendant de poids maximal et l'on distingue deux situations :

- 1er cas : si $s_k \in B$ le résultat est trivial.
- 2ème cas : si $s_k \notin B$. Remarquons que $\{s_k\}$ est un indépendant (car définition de s_k , s'il a été sélectionné c'est que l'algorithme a testé $\{s_k\} \in I$ or $\#\{s_k\} < \#B$ donc par échange on trouve $b_1 \in B$ tel que $\{s_k, b_1\} \in I$. On recommence $\#B - 1$ fois jusqu'à obtenir $\#\{s_k, b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_{\#B}\} = \#B$ avec b_i l'élément de B qui n'a pas été sélectionné dans le processus précédent. On pose $A' := B \setminus \{b_i\} \sqcup \{s_k\}$ qui est un indépendant (car) vérifiant

$$w(A') = w(B) - w(b_i) + w(s_k).$$

Or, $\{b_i\} \subset B$ donc $\{b_i\} \in I$ par hérédité, donc l'algorithme a considéré s_k avant b_i de sorte que $w(s_k) \geq w(b_i)$ et par conséquent

$$w(B) \leq w(A') \leq w(B) \quad (\text{par optimalité de } B).$$

D'où A' est optimal et contient s_k ce qu'on voulait démontrer. ■

Lemme A.5. Soit s_k le premier objet sélectionné par l'algorithme glouton. On pose $S' := S \setminus \{s_k\}$ et $I' := \{X \subset S' : X \cup \{s_k\} \in I\}$. (S', I') est un matroïde.

Démonstration. On vérifie les deux conditions de la définition :

- (Hérédité) Soit $Y \in I'$ puis $X \subset Y$. Par conséquent $X \subset S'$ car $Y \subset S'$ et donc $X \sqcup \{s_k\} \subset Y \sqcup \{s_k\}$. Par hypothèse $Y \sqcup \{s_k\} \in I$ et par hérédité dans (S, I) , on obtient $X \sqcup \{s_k\} \in I$ si bien que $X \in I'$.
- (Échange) Soit $(A, B) \in I'^2$ tel que $\#A < \#B$. Ainsi $\#(A \sqcup \{s_k\}) < \#(B \sqcup \{s_k\})$ avec $A \sqcup \{s_k\}$ et $B \sqcup \{s_k\}$ dans I , donc par échange dans (S, I) , il existe $x \in (B \sqcup \{s_k\}) \setminus (A \sqcup \{s_k\}) = B \setminus A$ tel que $(A \sqcup \{s_k\}) \sqcup \{x\} \in I$ si bien $A \sqcup \{x\} \in I'$. d'où le résultat. ■

On dispose maintenant des outils nécessaires pour conclure :

Théorème A.6. Cet algorithme donne un indépendant de cardinal maximum.

Démonstration. En premier lieu, puisque $\emptyset \in I$ d'après A.0, on détient immédiatement l'invariant de boucle " $A \in I$ " d'où l'on tire que le résultat de GLOUTON $((S, I))$ est bien un indépendant.

Montrons désormais que $\sum_{s \in A} w(s)$ est maximale ce que l'on montre par récurrence. Notons, pour tout $k \in \mathbb{N}^*$, H_k la propriété :

Pour tout matroïde (S, I) avec $\#S = k$ l'algorithme glouton renvoie un indépendant de cardinal maximum

- Pour $\#S = 1$ le résultat est évident.
- Soit $k \geq 2$ tel que H_{k-1} . On se donne (S, I) un matroïde avec $\#S = k$. L'algorithme glouton depuis $\{s_k\}$ donne un indépendant de poids maximal dans (S', I') par hypothèse de récurrence puisque $\#S' = \#S - 1 = k - 1$. Donc $\text{GROUTON}((S, I))$ renvoie un indépendant de poids maximal d'après le lemme précédent. ■

Corollaire A.7. *L'algorithme de Kruskal est correct*

Démonstration. D'après le premier exemple de ce court chapitre, les forêts d'un arbres forment un matroïde (c'est le premier exemple vu). L'algorithme Glouton appliqué à ce cas particulier donne exactement l'algorithme de Kruskal et l'on vient de voir qu'il est correct. ■

Annexe B

Machines de Turing

Sommaire

B.0	Définition	193
-----	----------------------	-----

Vous n'êtes qu'une machine de
TURING, l'univers n'est qu'une
immense machine de TURING,
nous vivons dans une machine de
TURING

F.Hatat

B.0 Définition

Définition B.1. Soit Σ un alphabet auquel on adjoint un symbole spécial noté B , l'ensemble $E := \Sigma^{\mathbb{Z}}$ dont les éléments sont appelés états et représentent des bandes de données. On munit ce dernier d'une fonction de transition

$$\delta : E \times \Sigma \longrightarrow E \times \Sigma \times \{\leftarrow, \rightarrow, \downarrow, \top, \perp\}$$

et d'un mot initial u_0 écrit sur sa bande.

Définition B.2. La machine a un mot écrit initialement sur sa bande. Si elle arrive sur \top , on dit qu'elle accepte ce mot. Dans les autres cas on dit qu'elle le refuse

E	Σ	E	Σ	Action
\circ	0	Δ	0	\rightarrow
Δ	0	Δ	0	\rightarrow
\circ	1	\circ	1	\rightarrow
Δ	1	\circ	1	\rightarrow
\circ	B	\circ	B	\perp
Δ	B	Δ	B	\top

Exemple. On pose $\Sigma = \{0, 1, \beta\}$ où β est l'état initial des cases pas écrites) Cette machine reconnaît les nombres pairs écrits en binaire avec le bit de poids fait à gauche.

Définition B.3 (Algorithme). Un algorithme est la réalisation de la fonction de transition d'une machine de TURING.

Par suite, on peut retrouver les différentes notions de complexité d'un algorithme.

- Lorsqu'il s'agit du temps, dans le pire cas, c'est le nombre maximal d'étapes nécessaires à la machine pour converger vers \top .
 - Lorsqu'il s'agit d'espace, c'est la taille de la bande nécessaire à la réalisation de l'algorithme.
- quand tu te donnes à fond et que ta nouvelle patronne te fait savoir qu'elle s'en fout et que ça ne c

Annexe C

CCS et le PI calcul

Sommaire

On se donne un ensemble de noms. On définit par induction les processus par :

$$P = ()|a.P|\bar{a}.P|P|P + P$$

Exemple. $a.\bar{b}.()|a.\bar{a}.c.(d.()) + (\bar{c}.())|a.b.()$

On définit **des** relations entre processus.

- Si $a \in \Sigma, a.P \xrightarrow{a} p$
- Si $a \in \Sigma, a.P \xrightarrow{\bar{a}} p$
- $a.P|\bar{a}.Q \xrightarrow{\tau} P|Q$

et toutes les extensions : Si $P \xrightarrow{w} P'$, alors $P|Q \xrightarrow{w} P'|Q, Q|P \xrightarrow{w} Q|P'$ et on quotiente les processus par :

- $Q_1 Q_2 = Q_2|Q_1$
- $(Q_1|Q_2)|Q_3 = Q_1|(Q_2|Q_3)$
- $Q = Q|() - > Q$

Exemple. P précédent. (oscar utilise la carte fx)

Les processus (noeuds) et les \rightarrow (arêtes). C'est un LTS (Labelled Transition System)

Quand peut-on dire que deux processus sont équivalents ?

- Idée 1 : Quand ils ont les mêmes traces (suites de \rightarrow) dans le LTS. Attention Nouvelle Règle : Si $P \rightarrow P', P + Q \rightarrow P'(P + Q = Q + P)$

Exemple. $P_1 = \bar{a}.(b + c), P_2 = \bar{a}.b + \bar{a}.c$ Insérer chemin tixz chelou.

Un processus P simule Q si $\forall Q' \xrightarrow{l} Q', \exists P' / P \xrightarrow{l} P'$ et P' simule Q'

Exemple. P_1 simule P_2 . Est ce que P_2 simule P_1 ?

$$P_1 \xrightarrow{\bar{a}} b + c \xrightarrow{c} .\text{Et}, P_2 \xrightarrow{\bar{a}} c \rightarrow$$

Définition C.1 (Bisimulation). On dit que P_1 et Q_1 sont bisimilaires, et on note $P_1 \approx Q_1$ lorsque

- $\forall P_1 \xrightarrow{l} P_2, \exists Q'_1$ tel que $Q_1 \xrightarrow{l} Q'_1$ et $P'_1 \approx Q'_1$
- $\forall Q_1 \xrightarrow{l} Q'_1, \exists P'_1$ tel que $P_1 \xrightarrow{l} P'_1$ et $P'_1 \approx Q'_1$

Exemple. Les machines à café ne sont pas similaires.

Remarque. Un automate \mathcal{A} qui se trouve dans l'état q . On eût définir $(\mathcal{A}, q) \xrightarrow{a \in \Sigma} (\mathcal{A}, \delta(q, a))$. On peut définir une bisimulation sur les automates.

Aussi, une heuristique consistante est admissible (la réciproque n'est pas vraie). En effet, si h est consistante, pour toute arête (u, v) nous avons $h(u) - h(v) \leq w(u, v)$. Soit un chemin quelconque $p = (v_0 = u, \dots, v_k = t)$, nous avons :

$$\begin{aligned}
 & w(p) \\
 = & \{ \text{Déf. du poids d'un chemin} \} \\
 & \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \\
 \geq & \{ \text{Consistance de } h \} \\
 & \sum_{i=0}^{k-1} h(v_i) - h(v_{i+1}) \\
 = & \{ \text{Arithmétique} \} \\
 & h(u) - h(t) \\
 = & \{ t \text{ est la cible} \} \\
 & h(u)
 \end{aligned}$$

En particulier, dans le cas d'un plus court chemin : $h(u) \leq \delta(u, t)$.

Annexe D

Déduction naturelle sur les types

Sommaire

D.0	Déduction des types en OCaml	197
D.1	Calcul avec le tiers exclu et continuations	199
D.1.1	Allégorie du diable	199
D.1.2	Continuations	199

Le tiers-exclu, c'est le diable!

F.Hatat

D.0 Déduction des types en OCaml

Dans le but de effectuer l'inférences de types en OCaml, on va modéliser les expressions suivantes par des formules logiques.

$$\begin{array}{l|l} (x,y) : 'a * 'b & ('a \wedge 'b) \\ \text{type sum} = \text{Left of 'a} \mid \text{Right of 'b} & \\ \text{Left}(x) & \\ \text{Right}(y) & ('a \vee 'b) \\ \text{fun } x \rightarrow y & ('a \rightarrow 'b) \end{array}$$

On note `fun x -> y` par $\lambda x.y$ dans le lambda calcul. On remarque que si on sait que `f` a le type `f : 'a ->'b` et `x : 'a` alors l'expression `f x` a nécessairement le type `'b`. On peut traduire cette constat par le règle suivant de déduction naturelle :

$$\frac{\Gamma \vdash f : 'a \rightarrow 'b \quad \Gamma \vdash x : 'a}{\Gamma \vdash f \ x : 'b} (e \rightarrow)$$

De manière similaire, si on sait que `y : 'b`, alors on peut déduire que `fun x->y : 'a->'b`.

$$\frac{\Gamma, x : 'a \vdash y : 'b}{\Gamma \vdash \text{fun } x \rightarrow y : 'a \rightarrow 'b} (i \rightarrow)$$

On dispose de règles suivantes pour le ou

$$\frac{\Gamma \vdash \text{Right}(x) : 'a \vee 'b}{\Gamma \vdash \text{Right}(x) : 'b} (i_d \vee) \quad \frac{\Gamma \vdash \text{Left}(x) : 'a \vee 'b}{\Gamma \vdash \text{Left}(x) : 'a} (i_g \vee)$$

Remarque. On note bien que les règles distingue la gauche du droite. La règle d'élimination du ou (non fourni ici) correspond au `match` de OCaml. Le contexte Γ , aussi appelé *environnement*, représente l'enregistrement des tous les types connues à un moment donné.

Les règles d'élimination de \wedge sont dérivable à partir du $\Gamma \vdash \text{fst} : 'a \wedge 'b \rightarrow 'a$ et $\Gamma \vdash \text{snd} : 'a \wedge 'b \rightarrow 'b$ qui traduisent le fait que **fst** et **snd** sont des fonctions du bibliothèque standard dont le type est connu.

$$\frac{\Gamma \vdash x : 'a \quad \Gamma \vdash y : 'b}{\Gamma \vdash (x, y) : 'a \wedge 'b} (i\wedge) \quad \frac{\Gamma \vdash x : 'a \wedge 'b}{\Gamma \vdash \text{fst } x : 'a} (e_g\wedge) \quad \frac{\Gamma \vdash x : 'a \wedge 'b}{\Gamma \vdash \text{snd } x : 'b} (e_d\wedge)$$

On a vu précédemment que fournir une preuve en déduction naturelle est en général un problème difficile, comment alors le compilateur d'OCaml peut s'en sortir ? La réponse est que le compilateur se contente de lire la preuve directement du code, et de le vérifier. En effet, la correspondance de Curry-Howard dit que il y a une bijection entre les preuves dans la logique intuitionniste et les programmes de lambda calculs typé. Le programme

fun (couple: 'a * 'a) -> fst couple

a le type $('a * 'a) \rightarrow 'a$ et en outre démontre $\vdash a \wedge a \rightarrow a$. On note que ici on a particulièrement intérêt de distinguer deux preuves distinctes de cette formule qui correspondes au 2 programmes différentes, ainsi que le choix des listes pour représenter le contexte au lieu des ensembles. Sans ses détails il y aura pas de bijection.

Exemple. Deux preuves et deux programmes

Formule	Programmes	Preuves
$a \wedge a \rightarrow a$	fun (couple : 'a * 'a) -> fst couple	$\frac{\frac{\frac{}{a \wedge a \vdash a \wedge a} (ax)}{a \wedge a \vdash a} (e_g\wedge)}{\vdash (a \wedge a) \rightarrow a} (i \rightarrow)$
	fun (couple : 'a * 'a) -> snd couple	$\frac{\frac{\frac{}{a \wedge a \vdash a \wedge a} (ax)}{a \wedge a \vdash a} (e_d\wedge)}{\vdash (a \wedge a) \rightarrow a} (i \rightarrow)$
$a \rightarrow (a \rightarrow a)$	fun (x:'a) -> fun (y:'a) -> x	$\frac{\frac{\frac{}{a, a \vdash a} (ax_0)}{a \vdash a \rightarrow a} (i \rightarrow)}{\vdash a \rightarrow (a \rightarrow a)} (i \rightarrow)$
	fun (x:'a) -> fun (y:'a) -> y	$\frac{\frac{\frac{}{a, a \vdash a} (ax_1)}{a \vdash a \rightarrow a} (i \rightarrow)}{\vdash a \rightarrow (a \rightarrow a)} (i \rightarrow)$

Le compilateur OCaml peut donc lire l'arbre de preuve directement du code, mais il connaît pas encore les types. A chaque feuille de l'arbre de preuve il associe un nouveau type fraîche. En procédant ainsi, il peut associer plusieurs types différent au même expression, il obtient un système de contraintes qui est résolu par l'algorithme de unification(cf le TP 12) pour enfin inférer le type de toute expression considéré. Si une solution existe, alors le programme est bien typé. Sinon, le programmeur a commis une erreur de type!

Exemple. On rappelle que $\lambda y.xy$ est une notation pour **fun x -> fun y -> x y**

$$\text{fun } x \rightarrow \text{fun } y \rightarrow x \ y \quad \frac{\frac{\frac{}{x, y \vdash x} (ax)}{x, y \vdash xy} (i \rightarrow)}{\vdash \lambda x. \lambda y. xy} (i \rightarrow)$$

Étape 0 : Code

Étape 1 : Arbre de preuve

$$\frac{\frac{\frac{\frac{}{x : 'a \rightarrow 'b, y : 'a \vdash x : 'a \rightarrow 'b} (ax)}{x : 'a \rightarrow 'b \vdash \text{fun } y \rightarrow x \ y : 'a \rightarrow 'b} (i \rightarrow)}{\vdash \text{fun } x \rightarrow \text{fun } y \rightarrow x \ y : ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b} (i \rightarrow)$$

Étape 2 : Inférence des types

Pour plus d'information sur ce sujet, voir page 680 du Turtle Book

D.1 Calcul avec le tiers exclu et continuations

D.1.1 Allégorie du diable

Le diable, dont le métier est d'offrir des pactes, vous approche avec une proposition. Sur son contrat, il est écrit : "Je te donne un milliard de dollars ou si tu me donne un milliard de dollars, je te donne ce que tu veux. Signer ici : _____." Contrairement à l'habitude, vous n'avez apparemment rien à perdre, donc vous signez hâtivement le pacte du diable. Le diable vous répond alors : "Donne moi un milliard de dollars et on en reparle." Pour autant, vous n'est pas Bernard Arnault, et vous n'avez pas un milliard de dollars.

Cette rencontre avec l'indivinité vous a marqué. À partir de ce moment, vous travaillez avec acharnement. Vous utilisez tous les moyens nécessaires, tout en vous souvenant de la promesse du diable. Après dix ans de salaire, contre toute attente vous amassez enfin 1 milliard de dollars.

Vous appelez le diable : "Voici le milliard de dollars, maintenant donne moi ce que je veux !" Le diable à cette instant même, ayant plus d'un tour dans son sac, rembobine le temps jusqu'à moment ou vous signez le pacte.

Vous n'avez apparemment rien à perdre, donc vous signez hâtivement le pacte du diable. Le diable vous répond alors : "Voici votre milliard de dollars."

D.1.2 Continuations

Cette petit histoire est une métaphore qui permet de comprendre la fonctionnent du tiers exclu dans cette modèle. Dans certains langages de programmation comme Scheme, la technique de continuation s'agit de deux fonctions : `save-cc` sauvegarde l'état de la mémoire, et `call-cc` qui charge un sauvegarde précédassent crée, effectivement permettant au programme de revenir en arrière dans le temps et changer les choix précédemment effectués. Les continuations permettent d'implémenter le principe du tiers-exclu ainsi :

```
let tiers a =  
  let m = save-cc in  
  Non(fun a -> call-cc m; Oui(a))
```

Un appelle à `tiers a` interroge sur la valeur de vérité de `a`. Le partie `fun a -> call-cc m` a le type `'a -> ⊥`, c'est à dire que "`a` est faux." Si on est capable de appeler cette fonction, alors on a une contradiction, donc on revient en arrière, en sachant que `a` est vrai. Donc la fonction tiers-exclue commence par dire que `a` est faux et revenir en arrière si il y a une contradiction. C'est un algorithme de retour sur trace.

Annexe E

Révisions

Sommaire

E.0	Branch and bound (évaluation et séparation)	201
E.1	SQL, Bases de données	203
E.1.1	Dans une base de données, il y a des tables	203
E.1.2	Rappels sur les requêtes	203
E.1.3	Correction de l'exercice KeyVentou	203

Moi j'ai pas besoin de révisions,
c'est pour mon voisin

Mon voisin

E.0 Branch and bound (évaluation et séparation)

Pour revenir sur le back tracking, on traite l'exercice sur le problème d'optimisation MAX-CLIQUE. Voici donc la correction

1. On l'appellera k -CLIQUE : Si G est non orienté, G contient-il une clique de taille au moins k ? On pose le certificat suivant :

On se donne la liste des sommets, et on vérifie que

- liste est de taille $\leq n$ et $\geq k$: $O(n \log n)$
- Tous les sommets entre 0 et $n - 1$: $O(n^3)$
- Deux à deux distincts : $O(n^2)$
- C'est bien une clique de G : $O(n^2)$ ou $O(n^3)$ en fonction de la représentation du graphe (matrice ou liste d'adjacence)

Ainsi, k -CLIQUE est NP. On va maintenant ramener ce problème à 3SAT :

Soit une formule de 3SAT : $C_1 \wedge \dots \wedge C_k$. $\forall i$, si $C_i = x_{i,1} * \vee x_{i,2} * \vee x_{i,3}$, on pose les sommets $x_{i,1}^*$, $x_{i,2}^*$, $x_{i,3}^*$ et on place une arête entre deux sommets $x_{i,j}^*$ et $x_{i',j'}^*$ si

$i \neq i'$ et si ce n'est pas la même variable avec des polarités différentes

Algorithme E.1 Max-Clique : retour sur trace

2. **Entrée :** Un graphe $G = (V, E)$ et une clique partielle P

Sortie : Le cardinal de la meilleure clique

```
fonction MAXCLIQUE( $P, G$ ) :  
    meilleure  $\leftarrow \#P$   
     $Q \leftarrow P$   
    Pour  $v \in G$  et  $v \notin P$ , faire :  
        Si  $P \sqcup \{v\}$  est une clique alors  
             $t, P' \leftarrow \text{MAXCLIQUE}(P \sqcup \{v\}, G)$   
            Si  $t > \text{meilleure}$  alors  
                meilleure  $\leftarrow t$   
                 $Q \leftarrow P$   
        Fin si  
    Fin si  
    renvoyer (meilleure,  $Q$ )  
Fin fonction
```

MAXCLIQUE(\square, G)

Étudions la complexité de cette bête là : attention c'est violent. On est sur du $O(n! \times \dots)$. En effet, quand il reste k sommets pas encore traités, on applique max-clique sur ces k sommets qui se fait en $O(k \times \text{max-clique sur les } k \text{ restants})$. Ainsi, si la plus grande clique est de taille k , l'algorithme la construit $k!$ fois... c'est très moche...

Ce n'est en fait pas très surprenant de par la nature du back-tracking : il est fait pour renvoyer la solution optimale, mais il n'est pas fait pour être efficace.

3. Le plaisir de cette question est laissé au lecteur.

Algorithme E.2 Max-clique : Bron Kerbosch

4. **Entrée :** Un graphe $G = (V, E)$ et une clique partielle P

Sortie : Le cardinal de la meilleure clique

```
fonction BK( $R, P, X$ ) :  
    Si  $P = \emptyset$  ( et  $X = \emptyset$  ) alors  
        renvoyer  $R$   
    Fin si  
    Si  $P \neq \emptyset$  alors  
        Pour  $v \in P$ , faire :  
             $P' \leftarrow \text{BK}(R \cup \{v\}, P \cap N(v), X \cap N(v))$   
             $P \leftarrow P \setminus \{v\}$   
             $X \leftarrow X \cup \{v\}$   
        Fin pour  
    Fin si  
    renvoyer le plus grand des  $P'$   
Fin fonction
```

5. Pour améliorer l'algo, on peut faire ceci : Soit $u \in P \cup X$.

Pour chaque $v \in P$

$N(u) : \text{Bk}(\dots)$

6. On a $\chi(G) \geq \omega(G)$

7. Si $G = (V, E)$, avec $|V| = n$, on a $\chi(G) \leq n$

8. Glouton donne $\tilde{\chi}(G)$. $\Delta(G) = \max$ des degrés. On a donc $\chi(G) \leq \Delta(G) + 1$

9. $\omega(G) \geq \tilde{\omega}(G)$. Glouton pour une clique maximale pour l'inclusion.

E.1 SQL, Bases de données

E.1.1 Dans une base de données, il y a des tables

Rappel de quelques concepts à maîtriser : Tables, clé étrangères, clés primaires, modèle entité-relation (ça, c'est de la théorie sur le SQL qui ne sert à rien).

E.1.2 Rappels sur les requêtes

- **SELECT** <attributs> **FROM** <tables> : C'est le **SELECT**, quoi. Argument * pour sélectionner toutes les colonnes. Avec plusieurs tables après le **FROM**, cela fait une table produit (c'est rare de le faire). Si on a des ambiguïtés avec plusieurs tables contenant les mêmes noms d'argument, on peut ajouter le nom de la table avant l'argument.
- **WHERE** <condition> : Garde les lignes vérifiant la condition demandée. Pas de fonction d'agrégation dans le **WHERE**.
- **JOIN** <table> **ON** <condition> : Cherche les correspondances, à l'inverse du produit cartésien. On peut mettre plusieurs conditions dans le **ON** avec les opérateurs **AND** et **OR**. On peut joindre une table à elle-même.
- **LEFT JOIN** <table> **ON** <condition> : Comme le **JOIN**, sauf pour les liens ne correspondant à aucune entrée dans la table demandée : on produit quand même la ligne correspondante, avec toutes les valeurs à **NULL**.
- **SUM**(<colonne>) : Additionne les valeurs de la colonne et la renvoie.
- **MAX**(<colonne>), **MIN**(<colonne>), **AVG**(<colonne>) : Maximum, minimum et moyenne des données.
- **COUNT**(<colonne>) : Compte le nombre d'entrées dans la colonne. Deux formes : **COUNT**(*) compte toutes les lignes, **COUNT**(colonne) ne compte que les lignes non nulles.
- **GROUP BY** <attribut> : Groupe les lignes ayant la même valeur pour l'attribut demandé.
- **HAVING** <fonction d'agrégation> : Effet similaire au **WHERE**, mais s'exécute en dernier, ce qui lui permet d'utiliser des fonctions d'agrégation. Doit être utilisé avec **GROUP BY**.
- **AS** <nom> : renomme aussi bien des colonnes que des tables pour se libérer des ambiguïtés.
- **ORDER BY** <int> **ASC** / **DESC** : Ordonne les données selon un attribut. Plusieurs attributs produisent un ordre lexicographique.
- **LIMIT** <int> : Affiche les n premières lignes. Doit être utilisé avec un **ORDER BY**.
- **DISTINCT** : A utiliser juste après le **SELECT** pour enlever les doublons.
- **UNION**, **INTERSECT**, **EXCEPT** : Fait l'union, l'intersection, la différence des tables résultantes. Il faut absolument que les tables aient les mêmes colonnes.

Remarque. (i) Le SQL est officiellement insensible à la casse : personne ne sait pourquoi on écrit tout en majuscule. Mais vous pouvez écrire **sELEct**, **select**, **SELECT**, **sELEcT**...

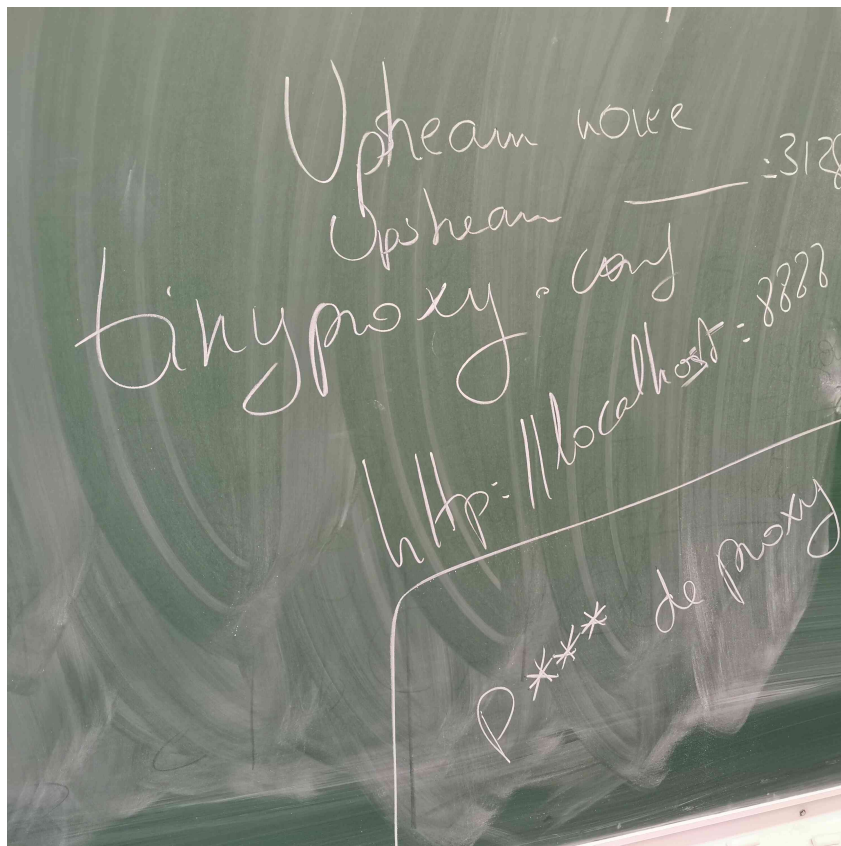
(ii) On peut effectuer des opérations arithmétiques entre colonnes si les types sont bien compatibles

(iii) Tout argument de type table peut être remplacé par une sous-requête. On peut aussi remplacer une donnée par une sous-requête renvoyant une unique case. On confond donc une table d'une ligne et d'une colonne avec une valeur, de la même manière qu'en mathématiques on confond $\mathcal{M}_1(\mathbb{K})$ et \mathbb{K} .

E.1.3 Correction de l'exercice KeyVentou

12. **SELECT** codeProduit **FROM** Produit **LEFT JOIN** Contient **ON** codeProduit = codeP **WHERE** codeP **IS NULL** **ORDER BY** codeProduit

Bonus. **SELECT** codeProduit **AS** code, **SUM**(prixHT * qte) **AS** valeur **FROM** Produit **AS** P **JOIN** ... **WHERE** ... **GROUP BY** codeProduit



Bonnes révisions et bons concours.