Outils Logiques

Roberto M. Amadio

Outils Logiques

Roberto M. Amadio
Université Paris-Diderot

24 novembre 2018

Table des matières

Préface

On trouve de nombreux livres, notes de cours, vidéos,... qui proposent une introduction adéquate à la logique. Dans ce sens ces notes de cours sont redondantes ; elles n'ont d'autre ambition que de fournir une *trace synthétique* des sujets traités dans un cours d'introduction aux *outils logiques* qui s'adresse aux étudiants d'informatique de l'Université Paris Diderot. [1] La lecture des premiers 6 chapitres de [BA12] est fortement conseillée. Le chapitre 2 de [GLM97] donne une perspective un peu plus avancée et les sections de [Knu12] et [Knu18] dédiées au calcul propositionnel sont une mine d'informations. Pour approfondir les notions traitées dans le chapitre 1 (Préliminaires), on peut lire [BN99] (chapitres 2 et 3).

La première partie du cours introduit le calcul propositionnel. On présente notamment la *syntaxe* et la *sémantique* du calcul et les notions de *validité*, *satisfaisabilité* et *équivalence logique*. On discute aussi un système de preuve basé sur le *calcul des séquents* et on aborde ainsi les concepts de *correction*, *complétude*, *conséquence logique* et *compacité*. La deuxième partie s'intéresse à 3 méthodes de déduction automatique qui s'appliquent au calcul propositionnel : la méthode de *résolution* pour la réfutation d'une formule, la *méthode de Davis-Putnam* pour la satisfaisabilité d'une formule et les *diagrammes de décision binaire* pour la représentation d'une fonction booléenne.

Du coté des pre-réquis *mathématiques*, on suppose que l'étudiant a déjà été exposé au raisonnement mathématique (par exemple, preuve par l'absurde, preuve par récurrence,...) et à la manipulation d'assertions semi formelles. On suppose aussi qu'il a été confronté à la présentation axiomatique de structures algébriques (groupe, anneau, corps,...). Coté *informatique*, on suppose que l'étudiant a une bonne connaissance d'un *langage de programmation*. En particulier, on suppose qu'il a été sensibilisé à la différence entre *syntaxe* et *sémantique* et qu'il sait manipuler les conditions logiques du langage à l'aide des opérateurs de conjonction, disjonction et négation. On suppose aussi que l'étudiant a été introduit à des *notions d'algorithmique* : structures de données élémentaires, arbres, graphes, notion de complexité asymptotique d'un algorithme dans le pire des cas, notation O. Une certaine familiarité avec les notions d'*automate fini* et de *circuit combinatoire* permettra d'apprécier l'intérêt de certaines digressions.

1. En rédigeant ces notes, j'ai bénéficié des remarques de mes collègues Ahmed Bouajjani, Antonio Bucciarelli et Michele Pagani ; bien entendu, les erreurs qui restent sont les miennes.

Notation

Ensembles

$\emptyset$	ensemble vide
$\mathbf{2} = \{0, 1\}$	valeurs booléennes
$\mathbf{N}$	nombres naturels
$\mathbf{Z}$	nombres entiers
$\mathbf{Z}_m = \{0, \ldots, m-1\}$	nombres entiers modulo m
$\mathbf{R}$	nombres réels
$\cup, \cap$	union, intersection de deux ensembles
$\bigcup, \bigcap$	union, intersection d'une famille d'ensembles
X^c	complémentaire de X
Y^X	fonctions de X dans Y
$\mathcal{P}(X)$	sous-ensembles de X
$\mathcal{P}_{\mathit{fin}}(X)$	sous-ensembles finis de X
$\sharp X$	cardinale de X
R^*	clôture réflexive et transitive d'une relation R

Logique (syntaxe)

$x, y, \ldots$	variables
$A, B, \ldots$	formules
$[B/x]A$	substitution d'une formule
$\wedge$	conjonction
$\vee$	disjonction
$\neg$	négation
$\rightarrow$	implication
$\leftrightarrow$	si et seulement si
$\oplus$	ou exclusif
$\rightsquigarrow$	conditionnel
$\beta, \beta', \ldots$	diagrammes de décision binaire

Logique (sémantique)

$v, v', \ldots$			affectations
$[\![A]\!]v$			interprétation d'une formule
$(v[b/y])(x) =$	$\begin{cases} b \\ v(x) \end{cases}$	si $x = y$ autrement	mise à jour d'une affectation

Algorithmique

f est $O(g)$	$\exists n_0, k \geq 0\ \forall n \geq n_0\ (f(n) \leq k \cdot g(n))$
f polynomiale	$\exists d \geq 0\ f$ est $O(n^d)$

Chapitre 1

Préliminaires

1.1 Algèbre initiale

Un zeste d'algèbre va nous permettre de préciser les notions de 'syntaxe' et de 'sémantique' (ou 'interprétation de la syntaxe').

Définition 1 (signature) *Une signature est un couple* (Σ, ar) *où* Σ *est un ensemble et* $ar : \Sigma \to \mathbf{N}$ *est un fonction qui associe à chaque élément de* Σ *un nombre naturel.*

On peut penser aux éléments de Σ comme à des *symboles* de fonction et à ar comme la fonction qui associe à chaque symbole de fonction son *arité*, c'est-à-dire le nombre d'arguments attendus par le symbole de fonction. Pour indiquer un symbole de fonction f avec $ar(f) = n$, on écrira aussi f^n.

Exemple 1 *Voici quelques exemples de signatures avec des dénominations qui seront justifiées dans la suite (exemple 4).*

$\{z^0, s^1\}$	*signature des nombres unaires*
$\{\epsilon^0, a^1, b^1, \ldots, z^1\}$	*signature des mots finis sur* $\{a, b, \ldots, z\}$
$\{nil^0, b^2\}$	*signature des arbres binaires (ordonnés et enracinés)*
$\{0^0, 1^0, +^2, ite^3, x^0, y^0, z^0, \ldots\}$	*signature des expressions conditionnelles numériques*
$\{\neg^1, \wedge^2, \vee^2, x^0, y^0, z^0, \ldots\}$	*signature des formules du calcul propositionnel.*

Définition 2 (Σ-algèbre) *Soit* (Σ, ar) *une signature. Une* Σ*-algèbre est composée d'un ensemble* A *et d'un ensemble de fonctions* $\{f_s : A^{ar(s)} \to A \mid s \in \Sigma\}$. *Dans une* Σ*-algèbre on a donc un ensemble et une fonction pour chaque symbole de la signature.*

Exemple 2 *Considérons la signature* $\Sigma = \{z^0, s^1\}$. *On peut construire une* Σ*-algèbre en prenant l'ensemble des nombres naturels avec la fonction constante* $f_z = 0$ *et la fonction unaire* $f_s(x) = x + 2$. *Une autre* Σ*-algèbre pourrait être l'ensemble des nombres réels avec la fonction constante* $g_z = 1$ *et la fonction unaire* $g_s(x) = 3 \cdot x$.

Remarque 1 *Si l'ensemble qui compose une* Σ*-algèbre est vide alors la signature* Σ *ne contient pas de symboles d'arité* 0.

Définition 3 (morphisme) *Soient* $(A, \{f_s \mid s \in \Sigma\})$ *et* $(B, \{g_s \mid s \in \Sigma\})$ *deux* Σ*-algèbres. On dit que la fonction* $h : A \to B$ *est un morphisme si elle commute avec les opérations de l'algèbre, à savoir pour tout* $s \in \Sigma$ *tel que* $ar(s) = n$ *et pour tout* $a_1, \ldots, a_n \in A$ *on a :*

$$h(f_s(a_1, \ldots, a_n)) = g_s(h(a_1), \ldots, h(a_n)) \ .$$

Exemple 3 *On reprend les deux* Σ*-algèbres de l'exemple 2. Le lecteur peut vérifier que la fonction* $h : \mathbf{N} \to \mathbf{R}$ *suivante est un morphisme :*

$$h(n) = \begin{cases} 3^k & \textit{si } n = 2 \cdot k \\ 0 & \textit{autrement.} \end{cases}$$

Exercice 1 *Trouvez un autre morphisme entre les* Σ*-algèbres introduites dans l'exemple 2.*

Il y a une façon *canonique* de construire une Σ-algèbre qu'on appelle Σ-algèbre *initiale*; cette dénomination est justifiée par la proposition 1.

Définition 4 (Σ-algèbre initiale) *Soit* (Σ, ar) *une signature. On définit les ensembles :*

$$\begin{array}{ll} T_0 & = \{s \in \Sigma \mid ar(s) = 0\}, \\ T_{n+1} & = T_n \cup \{(s, t_1, \ldots, t_m) \mid s \in \Sigma, ar(s) = m \geq 1, t_i \in T_n, i = 1, \ldots, m\}, \\ T_\Sigma & = \bigcup_{n \geq 0} T_n \ . \end{array}$$

Soit $s \in \Sigma$*. Si* $ar(s) = 0$ *on définit la (fonction) constante* $\underline{s} = s \in T_\Sigma$*. Et si* $ar(s) = m \geq 1$ *on définit une fonction* $\underline{s} : (T_\Sigma)^m \to T_\Sigma$ *par :*

$$\underline{s}(t_1, \ldots, t_m) = (s, t_1, \ldots, t_m) \ .$$

Exemple 4 *On explicite quelques éléments de l'ensemble* T_Σ *pour les signatures* Σ *introduites dans l'exemple 1.*

Signature	T_Σ
Nombres unaires	$\{z, (s, z), (s, (s, z)), (s, (s, (s, z))), \ldots\}$
Mots finis	$\{\epsilon, (a, \epsilon), \ldots, (z, \epsilon), (a, (a, \epsilon)), \ldots\}$
Arbres binaires	$\{nil, (b, nil, nil), (b, (b, nil, nil), nil), (b, nil, (b, nil, nil)), \ldots\}$
Expressions conditionnelles	$\{0, 1, x, \ldots, (+, 0, 0), \ldots, (ite, x, y, z), \ldots\}$
Formules propositionnelles	$\{x, \ldots, (\neg, x), \ldots, (\wedge, x, y), \ldots\}$

Proposition 1 *Soit* $\underline{A} = (A, \{f_s \mid s \in \Sigma\})$ *une* Σ*-algèbre. Alors il existe un unique morphisme de la* Σ*-algèbre initiale dans* $\underline{A}$*.*

Preuve. D'après la définition 4 d'algèbre initiale, pour tout $x \in T_\Sigma$ on peut définir :

$$rang(x) = min\{n \in \mathbf{N} \mid x \in T_n\} \ .$$

On montre que pour tout $x \in T_\Sigma$, il y a une seule définition possible de $h(x)$. On procède par récurrence sur $rang(x)$. Si $rang(x) = 0$ alors $x = s \in \Sigma$, $ar(s) = 0$ et on doit avoir $h(\underline{s}) = h(s) = f_s$. Si $rang(x) = n + 1$ alors $x = (s, t_1, \ldots, t_m)$, $ar(s) = m \geq 1$, $rang(t_i) \leq n$ pour $i = 1, \ldots, m$ et on doit avoir :

$$h(\underline{s}(t_1, \ldots, t_m)) = h(s, t_1, \ldots, t_m) = f_s(h(t_1), \ldots, h(t_m)) \ .$$

□

Exemple 5 *Pour définir un morphisme sur une Σ-algèbre initiale il suffit de fixer l'interprétation des symboles de la signature. Considérons les Σ-algèbres initiales de l'exemple 4. On peut définir la fonction qui associe un nombre naturel à un nombre en représentation unaire en considérant l'ensemble des nombres naturels avec fonctions $f_z = 0$ et $f_s(x) = x+1$. La longueur d'un mot correspond à la Σ-algèbre sur les nombres naturels où $f_\epsilon = 0$ et $f_a(x) = \cdots = f_z(x) = x + 1$. De façon similaire, la fonction qui calcule la hauteur d'un arbre binaire correspond à la Σ-algèbre sur les nombres naturels où $f_{nil} = 0$ et $f_b(x, y) = 1 + max(x, y)$. Pour construire un morphisme qui associe un nombre à chaque expression conditionnelle, on peut prendre l'ensemble des nombres naturels. On associe aux symboles 0 et 1 les nombres 0 et 1. On suppose aussi disposer d'une fonction $v : V \to \mathbf{N}$ des variables aux nombres naturels qui spécifie le nombre associé à chaque variable. Bien sûr, on interprète le symbole + par l'addition sur les nombres naturels et le symbole ite par la fonction ternaire :*

$$f_{ite}(x, y, z) = \begin{cases} y & si\ x \neq 0 \\ z & autrement. \end{cases}$$

De façon similaire, on peut associer une valeur à chaque formule du calcul propositionnel en prenant l'ensemble des valeurs booléennes $\mathbf{2} = \{0, 1\}$. On suppose disposer d'une fonction $v : V \to \mathbf{2}$ des variables aux valeurs booléennes qui spécifie la valeur de chaque variable. Il reste à définir les fonctions NOT, AND et OR associées aux symboles $\neg$, $\wedge$ et $\vee$, respectivement. Ces fonctions sont spécifiées par les tableaux suivants :

x	$NOT(x)$
0	1
1	0

x	y	$AND(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$OR(x,y)$
0	0	0
0	1	1
1	0	1
1	1	1

En changeant de Σ-algèbre, on peut trouver d'autres interprétations intéressantes des formules propositionnelles. Par exemple, on peut associer à chaque formule sa taille, à savoir le nombre de symboles qu'elle contient. Dans ce cas, on prend comme domaine d'interprétation l'ensemble des nombres naturels, on associe aux variables le nombre naturel 1 et on définit les fonctions $f_\neg(x) = x + 1$ et $f_\wedge(x, y) = f_\vee(x, y) = 1 + x + y$. Dans une autre direction, on peut fixer un ensemble E et interpréter les variables comme des sous-ensembles de E et les symboles $\neg$, $\wedge$ et $\vee$ comme le complémentaire, l'intersection et l'union, respectivement.

1.2 Définitions inductives

On trouve souvent des définitions dites 'inductives' de la forme 'le plus petit ensemble qui satisfait un certain nombre de propriétés'. On donne une condition suffisante pour l'existence d'un tel ensemble et on montre comment manipuler ce type de définitions.

Définition 5 (fonction monotone) *Soit E un ensemble et $\mathcal{P}(E)$ les parties de E. On dit qu'une fonction $\mathcal{F} : \mathcal{P}(E) \to \mathcal{P}(E)$ est* monotone *si pour tout $X \subseteq Y \subseteq E$ on a $\mathcal{F}(X) \subseteq \mathcal{F}(Y)$.*

Proposition 2 *Soient E un ensemble et $\mathcal{F} : \mathcal{P}(E) \to \mathcal{P}(E)$ une fonction monotone. Alors :*

$$Y = \bigcap \{X \mid X \subseteq E \text{ et } \mathcal{F}(X) \subseteq X\} ,$$

est le plus petit ensemble $X \subseteq E$ tel que $\mathcal{F}(X) \subseteq X$.

PREUVE. Si $\mathcal{F}(X) \subseteq X$ alors par définition de Y on a $Y \subseteq X$. Il reste à montrer que $\mathcal{F}(Y) \subseteq Y$. On observe :

$$\begin{array}{ll} \mathcal{F}(Y) & = \mathcal{F}(\bigcap\{X \mid X \subseteq E \text{ et } \mathcal{F}(X) \subseteq X\}) \\ & \subseteq \bigcap\{\mathcal{F}(X) \mid X \subseteq E \text{ et } \mathcal{F}(X) \subseteq X\} \quad \text{(par monotonie)} \\ & \subseteq \bigcap\{X \mid X \subseteq E \text{ et } \mathcal{F}(X) \subseteq X\} \\ & = Y \ . \end{array}$$

□

Remarque 2 *Dans les hypothèses de la proposition 2, pour montrer que $Y \subseteq X$, il suffit de montrer que $\mathcal{F}(X) \subseteq X$.*

Exemple 6 *Soit A un ensemble et $R \subseteq A \times A$ une relation binaire sur A. On dit que la relation R est réflexive si pour tout $x \in A$ on a que $(x, x) \in R$ et qu'elle est transitive si pour tout $x, y, z \in A$ elle satisfait la condition :*

$$(x, y) \in R \text{ et } (y, z) \in R \text{ implique } (x, z) \in R \ .$$

On est intéressé à la plus petite relation réflexive et transitive qui contient R. Vérifions que l'existence d'une telle relation suit de la proposition 2. Prenons $E = A \times A$. Soit $Id_A = \{(x, x) \mid x \in A\}$ la relation identité. On définit $\mathcal{F} : \mathcal{P}(E) \to \mathcal{P}(E)$ par :

$$\mathcal{F}(S) = Id_A \cup R \cup \{(x, z) \mid \exists\, y \ \ (x, y) \in S \text{ et } (y, z) \in S\} \ .$$

On remarque que $S \subseteq S'$ implique $\mathcal{F}(S) \subseteq \mathcal{F}(S')$, donc la fonction $\mathcal{F}$ est monotone. Aussi une relation S contient la relation R et est réflexive et transitive ssi $\mathcal{F}(S) \subseteq S$. On peut donc conclure que la plus petite relation transitive qui contient R existe et est égale à :

$$\bigcap\{S \mid S \subseteq E \text{ et } \mathcal{F}(S) \subseteq S\} \ .$$

On dénote cette plus petite relation par R^*. Considérons maintenant la relation T définie de la façon suivante :

$$\begin{array}{ll} T_0 & = Id_A \cup R \ , \\ T_{n+1} & = \{(x, z) \mid \exists\, y \ \ (x, y) \in T_n \text{ et } (y, z) \in T_n\} \ , \\ T & = \bigcup_{n \in \mathbf{N}} T_n \ . \end{array}$$

On peut montrer par récurrence sur n que $T_n \subseteq R^*$. Donc, par définition de T, on a $T \subseteq R^*$. D'autre part, pour montrer que $R^* \subseteq T$ il suffit de vérifier que $\mathcal{F}(T) \subseteq T$ et d'appliquer la remarque 2.

Exercice 2 *Soit R un relation comme dans l'exemple 6 . Montrez l'existence de la plus petite relation transitive qui contient R. On dénote cette relation par R^+.*

Exercice 3 *Montrez que l'ensemble Y de la proposition 2 est le plus petit point fixe de la fonction $\mathcal{F}$, à savoir le plus petit ensemble $X \subseteq E$ tels $\mathcal{F}(X) = X$.*

Exercice 4 *Soit Σ la signature des mots finis (exemple 1) et T_Σ l'ensemble associé à l'algèbre initiale (définition 4). Soit $\Sigma' = \{\epsilon^0, a^1, b^1\}$ une sous-signature de Σ. Montrez que $T_{\Sigma'}$ peut être défini comme un ensemble inductif sur T_Σ.*

1.3 Ordres bien fondés, principe d'induction et terminaison

On suppose que le lecteur est familier avec le principe de preuve par récurrence. Ce principe se généralise à un *principe d'induction* sur des structures plus générales que les nombres naturels qu'on appelle *ordres bien fondés*. Par ailleurs, il se trouve que les ordres bien fondés sont un outil logique naturel pour montrer la *terminaison* des programmes.

Définition 6 (ordre partiel) *Un ordre partiel $(P, >)$ est un ensemble P avec une relation binaire $> \subseteq P \times P$ qui est transitive. On utilisera aussi $\geq$ comme un abréviation pour la relation binaire $(> \cup Id_P)$.*

Définition 7 (ordre partiel bien fondé) *Un ordre partiel $(P, >)$ est bien fondé s'il n'existe pas une séquence $\{x_i\}_{i \in \mathbf{N}}$ dans P telle que $x_0 > x_1 > x_2 > \cdots$.*

Remarque 3 *Dans un ordre partiel bien fondé, on ne peut pas avoir un x tel que $x > x$ car dans ce cas la suite où $x_i = x$ pour tout $i \in \mathbf{N}$ contredit la condition de bonne fondation. Dans cette section, il convient de penser à la relation $>$ comme à un ordre strict.*

Exemple 7 *Voici des exemples d'ordres bien fondés : les nombres naturels avec l'ordre standard, les mots finis ordonnés d'après leur longueur, les arbres binaires ordonnés d'après leur hauteur, les formules du calcul propositionnel ordonnées d'après leur taille. Par contre, les nombres entiers avec l'ordre standard ainsi que les nombres rationnels positifs avec l'ordre standard ne sont pas des ordres bien fondés.*

Exercice 5 *Soient $\mathbf{N}$ l'ensemble des nombres naturels, $\mathbf{N}^k$ le produit cartésien $\mathbf{N} \times \cdots \times \mathbf{N}$ k fois et $A = \bigcup\{\mathbf{N}^k \mid k \geq 1\}$. Soit $>$ une relation binaire sur A telle que : $(x_1, \ldots, x_n) > (y_1, \ldots, y_m)$ ssi il existe $k \leq min(n, m)$ $(x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, x_k > y_k)$. L'ordre $>$ est-il bien fondé ?*

Définition 8 (principe d'induction) *Soit $(P, >)$ un ordre partiel. Si $x \in P$ on dénote par $\downarrow(x) = \{y \in P \mid x > y\}$ l'ensemble des éléments plus petits que x. Soit $X \subseteq P$. Le principe d'induction affirme que pour montrer que $X = P$ il suffit de montrer que pour tout $x \in P$ si $\downarrow(x) \subseteq X$ alors $x \in X$.*

On peut voir le principe d'induction comme une généralisation du principe de récurrence sur les nombres naturels.

Proposition 3 *Si $(P, >)$ est un ordre partiel bien fondé alors le principe d'induction est valide.*

Preuve. Soit $X \subseteq P$ un ensemble qui satisfait le principe d'induction, à savoir : pour tout $x \in P$ si $\downarrow(x) \subseteq X$ alors $x \in X$. Soit $x_0 \in P$. Si $\downarrow(x_0) = \emptyset$ alors le principe d'induction stipule que $x_0 \in X$. Donc supposons $x_0 \in P \backslash X$ et $\downarrow(x_0) \neq \emptyset$. On peut donc trouver $x_1 \in \downarrow(x_0)$ tel que $x_1 \in P \backslash X$. En effet si $\downarrow(x_0) \subseteq X$ alors le principe d'induction garantit que $x_0 \in X$. On peut maintenant itérer le même argument sur $x_1, x_2, \ldots$ On construit donc une séquence décroissante $x_0 > x_1 > x_2 > \cdots$ qui est en contradiction avec l'hypothèse que $(P, >)$ est bien fondé. □

Exercice 6 *Montrez que si $(P, >)$ est un ordre partiel dans lequel le principe d'induction est valide alors $(P, >)$ est bien fondé.*

Donc les ordres partiels bien fondés sont exactement les ordres partiels dans lesquels le principe d'induction est valide. On introduit deux méthodes pour construire un ordre bien fondé à partir d'ordres bien fondés.

Définition 9 (ordre produit) *Soient $(P_1, >_1)$ et $(P_2, >_2)$ deux ordres partiels. On définit l'ordre produit par $(P_1 \times P_2, >_{prod})$ où :*

$$(x, y) >_{prod} (x', y') \text{ si } (x >_1 x' \text{ et } y \geq_2 y') \text{ ou } (x \geq_1 x' \text{ et } y >_2 y') \ .$$

Proposition 4 *Si $(P_1, >_1)$ et $(P_2, >_2)$ sont deux ordres partiels alors $(P_1 \times P_2, >_{prod})$ est un ordre partiel. De plus si $(P_1, >_1)$ et $(P_2, >_2)$ sont deux ordres partiels bien fondés alors $(P_1 \times P_2, >_{prod})$ est bien fondé aussi.*

Preuve. On laisse au lecteur la vérification que $>_{prod}$ est transitif. Pour la bonne fondation, on procède par contradiction. Si $(x_0, y_0) >_{prod} (x_1, y_1) >_{prod} \cdots$ alors ou bien la première ou bien la deuxième composante diminue strictement infiniment souvent. On a donc une suite décroissante ou bien dans P_1 ou bien dans P_2. □

Définition 10 (ordre lexicographique) *Soient $(P_1, >_1)$ et $(P_2, >_2)$ deux ordres partiels. On définit l'ordre lexicographique par $(P_1 \times P_2, >_{lex})$ où :*

$$(x, y) >_{lex} (x', y') \text{ si } (x >_1 x') \text{ ou } (x = x' \text{ et } y >_2 y') \ .$$

Proposition 5 *Si $(P_1, >_1)$ et $(P_2, >_2)$ sont deux ordres partiels alors $(P_1 \times P_2, >_{lex})$ est un ordre partiel. De plus si $(P_1, >_1)$ et $(P_2, >_2)$ sont deux ordres partiels bien fondés alors $(P_1 \times P_2, >_{lex})$ est bien fondé aussi.*

Preuve. Encore une fois, la vérification de la transitivité est laissée au lecteur. Pour vérifier la bonne fondation, on procède aussi par contradiction. Si $(x_0, y_0) >_{lex} (x_1, y_1) >_{lex} \cdots$ alors ou bien la première composante diminue strictement infiniment souvent ou bien la première composante se stabilise et la deuxième diminue strictement. Dans le deux cas on a une contradiction. □

Remarque 4 *L'ordre produit et l'ordre lexicographique sont définis sur le produit cartésien et l'ordre produit implique toujours l'ordre lexicographique, à savoir : si $(x, y) >_{prod} (x', y')$ alors $(x, y) >_{lex} (x', y')$.*

On introduit un notion très générale de système de réécriture.

Définition 11 (système de réécriture) *Un système de réécriture est un couple $(X, \rightarrow)$ où X est un ensemble et $\rightarrow \subseteq X \times X$ est une relation binaire sur X.*

Définition 12 (terminaison) *Un système de réécriture $(X, \rightarrow)$ termine*[1] *s'il n'existe pas une séquence $\{x_i\}_{i \in \mathbf{N}}$ dans X telle que $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots$*

1. Tout problème de terminaison de programmes peut être formulé comme un problème de terminaison d'un système de réécriture.

Notez que si un système termine on ne peut pas avoir un x tel que $x \to x$. On dénote par $\overset{+}{\to}$ la clôture transitive de la relation $\to$ (exemple 6). Les ordres bien fondés et les systèmes de réécriture qui terminent sont deux faces de la même pièce.

Proposition 6 *Soit $(X, \to)$ un système de réécriture. Les assertions suivantes sont équivalentes.*

1. *$(X, \to)$ termine.*
2. *$(X, \overset{+}{\to})$ est un ordre partiel bien fondé.*
3. *Il existe un ordre partiel bien fondé $(P, >)$ et une fonction $\mu : X \to P$ telle que pour tout $x, y \in X$ si $x \to y$ alors $\mu(x) > \mu(y)$.*

PREUVE. (1) $\Rightarrow$ (2). Par définition, $\overset{+}{\to}$ est transitive et donc $(X, \overset{+}{\to})$ est un ordre partiel. Si on a une séquence $x_0 \overset{+}{\to} x_1 \overset{+}{\to} x_2 \overset{+}{\to}$ ou a aussi une séquence $x_0 \to \cdots \to x_1 \to \cdots \to x_2 \to \cdots$ ce qui contredit la terminaison.

(2) $\Rightarrow$ (3). Il suffit de prendre $(P, >) = (X, \overset{+}{\to})$ et $\mu = id_X$, la fonction identité sur X.

(3) $\Rightarrow$ (1). Si $x_0 \to x_1 \to x_2 \to \cdots$ alors on a $\mu(x_0) > \mu(x_1) > \mu(x_2) > \cdots$ ce qui contredit la bonne fondation de P. □

Exercice 7 *Considérons l'ensemble des mots finis sur les symboles $\{a, b\}$. On écrit $w \to w'$ si w' est obtenu de w en remplaçant un sous-mot ab par le mot bba. Trouvez une interprétation sur les nombres naturels positifs qui montre que $\to$ termine.*

Exercice 8 *Considérons la plus petite relation $\to$ sur $\mathbf{N} \times \mathbf{N}$ telle que pour tout $x, y, z \in \mathbf{N}$ on a que :*

$$(x + 1, y) \to (x, z) \ , \quad \text{et} \quad (x, y + 1) \to (x, y) \ .$$

Le système $(\mathbf{N} \times \mathbf{N}, \to)$ termine-t-il ?

Exercice 9 *Les programmes* while *suivants terminent-t-ils en supposant que les variables varient sur les nombres naturels positifs ?*

$$\mathsf{while}(m \neq n)\{\ \mathsf{if}(m > n)\{m = m - n;\}\ \mathsf{else}\ \{n = n - m;\}\ \}\ ,$$
$$\mathsf{while}(m \neq n)\{\ \mathsf{if}(m > n)\{m = m - n;\}\ \mathsf{else}\ \{h = m; m = n; n = h;\}\ \}\ .$$

Exercice 10 *Soient Σ la signature des formules du calcul propositionnel (exemple 1) et T_Σ l'ensemble associé à l'algèbre initiale (définition 4). Soit $\to$ la plus petite relation binaire sur T_Σ telle que pour tout $A, A', B \in T_\Sigma$:*

- $(\neg, (\neg, A)) \to A$,
- $(\neg, (\wedge, A, B)) \to (\vee, (\neg, A), (\neg, B))$,
- $(\neg, (\vee, A, B)) \to (\wedge, (\neg, A), (\neg, B))$,
- *si $A \to A'$ alors $(\neg, A) \to (\neg, A')$, $(\wedge, A, B) \to (\wedge, A', B)$, $(\wedge, B, A) \to (\wedge, B, A')$, $(\vee, A, B) \to (\vee, A', B)$ et $(\vee, B, A) \to (\wedge, B, A')$.*

On peut voir $(T_\Sigma, \to)$ comme un système de réécriture qui transforme une formule jusqu'à ce que les négations paraissent uniquement devant les variables. On dit qu'une formule avec cette propriété est en forme normale négative.

1. *Expliquez pourquoi la relation $\to$ existe.*

2. *Une formule A est en* forme normale *s'il n'existe pas A' tel que $A \to A'$. Donnez une description inductive de l'ensemble des formules en forme normale et de l'ensemble des formules qui ne sont pas en forme normale.*
3. *Montrez la terminaison du système de réécriture $(T_\Sigma, \to)$. Suggestion : interprétez les symboles comme des fonctions affines.*
4. *Montrez que si $A \overset{*}{\to} B$ et B est en forme normale alors la taille de B (exemple 5) est au plus 2 fois la taille de A.*

Chapitre 2

Calcul Propositionnel

2.1 Syntaxe et sémantique

La *logique* est à l'origine une réflexion sur le discours (*logos*) et sur sa cohérence. En particulier, la logique *mathématique* s'intéresse à l'organisation et à la cohérence du discours mathématique et donc aux notions de *validité* et de *preuve*. Dans le *calcul propositionnel classique*, on dispose d'un certain nombre de *propositions* qui peuvent être vraies ou fausses et d'un certain nombre d'opérateurs qui permettent de combiner ces propositions.

On rappelle (section 1.1) que la syntaxe du calcul propositionnel est définie comme l' algèbre initiale sur la signature $\Sigma = V \cup \{\neg^1, \wedge^2, \vee^2\}$ où $V = \{x^0, y^0, \ldots\}$ est un un ensemble dénombrable de symboles de *variables* d'arité 0. [1] On appelle *formules* les éléments de T_Σ et on les dénote par les lettres $A, B, \ldots$ On choisit une variable x_0 (arbitraire mais fixée) et on utilise les abréviations suivantes :

$$\begin{array}{c} \neg A = (\neg, A), \qquad (A \wedge B) = (\wedge, A, B), \qquad (A \vee B) = (\vee, A, B), \\ \mathbf{1} = (x_0 \vee \neg x_0), \qquad \mathbf{0} = (x_0 \wedge \neg x_0)\ . \end{array} \tag{2.1}$$

Définition 13 (littéral) *Un* littéral *est une formule qui est une variable ou la négation d'une variable. Dans le premier cas on dit que le littéral est* positif *et dans le deuxième qu'il est* négatif. *On dénote un littéral avec* $\ell, \ell', \ldots$

Définition 14 (variables dans une formule) *L'ensemble* $var(A)$ *des variables présentes dans une formule* A *est défini par :*

$$var(x) = \{x\}, \quad var(\neg A) = var(A), \quad var(A \wedge B) = var(A \vee B) = var(A) \cup var(B).$$

Définition 15 (substitution) *Si* A, B *sont des formules et* x *est une variable alors on dénote par* $[B/x]A$ *la substitution de la variable* x *par la formule* B *dans la formule* A. *La fonction de substitution est définie sur* T_Σ *par :*

$$[B/x](y) = \begin{cases} B & si\ y = x \\ y & autrement \end{cases} \qquad [B/x](\neg A) = \neg[B/x]A,$$

$$[B/x](A \wedge A') = ([B/x]A \wedge [B/x]A') \qquad [B/x](A \vee A') = ([B/x]A \vee [B/x]A')\ .$$

1. La *syntaxe*, telle qu'on l'entend dans ce cours, est aussi appelée *syntaxe abstraite* dans le cadre de l'analyse syntaxique des langages. Un programme d'analyse syntaxique reçoit en entrée une suite de caractères et produit en sortie soit la syntaxe abstraite d'une phrase du langage soit un message d'erreur.

Exercice 11 *Explicitez les Σ-algèbres qui induisent au sens de la proposition 1, la fonction var de la définition 14 et la fonction $[B/x](\_)$ de la définition 15 (voir exemple 5).*

On verra dans la section 2.3 que les opérateurs $\neg$, $\vee$ et $\wedge$ suffisent à exprimer tous les autres. Cependant, un certain nombre d'opérateurs logiques sont utilisés assez souvent pour mériter un symbole spécifique.

Définition 16 (opérateurs dérivés)

$$\begin{array}{lll} (A \to B) & = (\neg A \vee B) & \textit{(implication)} \\ (A \leftrightarrow B) & = ((A \to B) \wedge (B \to A)) & \textit{(si et seulement si)} \\ (A \oplus B) & = ((A \wedge \neg B) \vee (\neg A \wedge B)) & \textit{(ou exclusif)} \\ (A \rightsquigarrow B, C) & = ((A \wedge B) \vee (\neg A \wedge C)) & \textit{(conditionnel).} \end{array}$$

L'interprétation standard des formules utilise les *valeurs booléennes* $\mathbf{2} = \{0, 1\}$ avec la convention que 0 correspond à 'faux' et 1 à 'vrai'. On rappelle (section 1.1) qu'une fois qu'on a fixé l'interprétation des variables et des symboles $\neg$, $\vee$ et $\wedge$, on a une fonction qui est définie de façon unique.

Définition 17 (interprétation) *L'interprétation d'une formule A par rapport à une affectation $v : V \to \mathbf{2}$ est la fonction (unique) $[\![\_]\!]v$ qui satisfait :*

$$\begin{array}{ll} [\![x]\!]v = v(x) \ , & [\![\neg A]\!]v = NOT([\![A]\!]v) \ , \\ [\![A \wedge B]\!]v = AND([\![A]\!]v, [\![B]\!]v) \ , & [\![A \vee B]\!]v = OR([\![A]\!]v, [\![B]\!]v) \ , \end{array}$$

où les fonctions NOT, AND, OR sont définies par :

x	$NOT(x)$
0	1
1	0

x	y	$AND(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$OR(x,y)$
0	0	0
0	1	1
1	0	1
1	1	1

Définition 18 (mise à jour) *Si v est une affectation, x est une variable propositionnelle et $b \in \mathbf{2}$ est une valeur booléenne alors la mise à jour de v avec b pour x est définie par :*

$$v[b/x](y) = \begin{cases} b & \textit{si } y = x \\ v(y) & \textit{autrement.} \end{cases}$$

Notation On peut expliciter les valeurs d'une affectation en écrivant $[b_1/x_1, \ldots, b_n/x_n]$ où $x_i \neq x_j$ si $i \neq j$. Dans ce cas, il est entendu que les valeurs de l'affectation sur les variables différentes de $x_1, \ldots, x_n$ n'ont pas d'importance.

La proposition suivante met en relation la substitution au niveau syntaxique avec la mise à jour au niveau sémantique. Elle permet de remplacer les variables propositionnelles par des formules arbitraires.

Proposition 7 *Soient A, B deux formules, x une variable et v une affectation. Alors :*

$$[\![[B/x]A]\!]v = [\![A]\!]v[[\![B]\!]v/x] \ .$$

PREUVE. Par récurrence sur la *taille* (exemple 5) de A. □

On introduit maintenant 3 définitions fondamentales pour la suite du cours. On écrit $v \models A$ si $[\![A]\!]v = 1$ et $\models A$ si pour tout v, $v \models A$.

Définition 19 (validité) *Une formule A est valide (on dit aussi qu'elle est une* tautologie*) si $\models A$, c'est-à-dire si pour toute affectation v on a $v \models A$.*

Définition 20 (satisfaisabilité) *Une formule A est satisfaisable s'il existe une affectation v telle que $v \models A$.*

Définition 21 (équivalence logique) *Deux formules A et B sont logiquement équivalentes si pour toute affectation v on a $v \models A$ ssi $v \models B$. Dans ce cas on écrit $A \equiv B$.*

Par exemple, $\mathbf{1}$ est valide, x est satisfaisable mais pas valide, $\mathbf{0}$ n'est pas satisfaisable, x est équivalente à $x \wedge \mathbf{1}$ et n'est pas équivalente à $x \vee \mathbf{1}$. D'un point de vue mathématique, on peut prendre une de ces notions comme fondamentale et dériver les deux autres. [2]

Exercice 12 *Soient A, B deux formules du calcul propositionnel. Montrez que :*

1. *A est valide ssi $\neg A$ n'est pas satisfaisable,*
2. *A est valide ssi $A \equiv \mathbf{1}$,*
3. *$A \equiv B$ ssi $A \leftrightarrow B$ est valide.*

Même si les notions de validité, satisfaisabilité et équivalence logique sont d'une certaine façon interchangeables on verra dans la suite du cours que chaque notion a ses propres méthodes algorithmiques.

2.2 Équivalence logique

Proposition 8 *Dans le calcul propositionnel, on dispose des équivalences logiques présentées dans la table 2.1.*

PREUVE. En calculant les tables de vérité. □

Le groupe $(G1)$ dit que le $\vee$ est un opérateur commutatif, associatif et idempotent qui a $\mathbf{0}$ comme identité et $\mathbf{1}$ comme absorbant. Le groupe $(G2)$ dit que le $\wedge$ est aussi un opérateur commutatif, associatif et idempotent mais son identité est $\mathbf{1}$ et son absorbant est $\mathbf{0}$. Le groupe $(G3)$ permet de distribuer l'opérateur $\wedge$ ($\vee$) par rapport à l'opérateur $\vee$ ($\wedge$). Le groupe $(G4)$ affirme que la négation est un opérateur involutif et qu'il peut être distribué d'une certaine façon sur les opérateurs $\vee$ et $\wedge$ (lois de De Morgan). Avec ces équivalences on peut déduire $\neg\mathbf{0} \equiv \mathbf{1}$ et $\neg\mathbf{1} \equiv \mathbf{0}$. D'après la convention (2.1), les formules $\mathbf{1}$ et $\mathbf{0}$ dépendent d'une variable x_0 *fixée*. Le dernier groupe $(G5)$, nous permet de déduire, par exemple, que pour *toute* variable x, $(x \vee \neg x) \equiv \mathbf{1}$, aussi connue comme *loi du tiers exclu*.

2. Quand on traite des formules, attention à ne pas confondre la relation $=$ (identité) avec la relation $\equiv$ (équivalence logique) ; la première est strictement contenue dans la deuxième.

$$
\begin{array}{ll}
(G1) & (x \vee \mathbf{0}) \equiv x, \quad (x \vee \mathbf{1}) \equiv \mathbf{1}, \quad (x \vee y) \equiv (y \vee x), \\
& ((x \vee y) \vee z) \equiv (x \vee (y \vee z)), \quad (x \vee x) \equiv x, \\
(G2) & (x \wedge \mathbf{0}) \equiv \mathbf{0}, \quad (x \wedge \mathbf{1}) \equiv x, \quad (x \wedge y) \equiv (y \wedge x), \\
& ((x \wedge y) \wedge z) \equiv (x \wedge (y \wedge z)), \quad (x \wedge x) \equiv x, \\
(G3) & (x \wedge y) \vee z \equiv (x \vee z) \wedge (y \vee z), \quad (x \vee y) \wedge z \equiv (x \wedge z) \vee (x \wedge z), \\
(G4) & \neg\neg x \equiv x, \quad \neg(x \vee y) \equiv (\neg x \wedge \neg y), \quad \neg(x \wedge y) \equiv (\neg x \vee \neg y)\ , \\
(G5) & (x \vee \neg x) \equiv \mathbf{1}, \quad (x \wedge \neg x) \equiv \mathbf{0}.
\end{array}
$$

TABLE 2.1 – Équivalences logiques

Grâce à la proposition 7, on peut toujours remplacer une variable par une formule. Ainsi, si l'on veut montrer $(A \wedge A) \equiv A$ pour une formule A arbitraire, on fait appel à l'équivalence $(x \wedge x) \equiv x$ (groupe $(G2)$) et on note que pour toute affectation v :

$$[\![A \wedge A]\!]v \;\; = [\![[A/x](x \wedge x)]\!]v \;\; = [\![x \wedge x]\!]v[[\![A]\!]v/x] \;\; = [\![x]\!]v[[\![A]\!]v/x] \;\; = [\![A]\!]v\ .$$

Il est donc possible de pratiquer un *raisonnement équationnel* sur les formules du calcul propositionnel qui est similaire à celui que le lecteur a déjà pratiqué dans le cadre, par exemple, de la théorie des groupes. En effet, les équivalences de la table 2.1 forment la base d'une théorie équationnelle qu'on appelle *algèbre booléenne*.

Exercice 13 *Soient A, B, C, D des formules x une variable. Montrez que si $A \equiv B$ et $C \equiv D$ alors $[A/x]C \equiv [B/x]D$.*

Exercice 14 *Utilisez un raisonnement équationnel pour déduire les (célèbres) lois suivantes :*

$$
\begin{array}{ll}
((\neg y \to \neg x) \to (x \to y)) \equiv \mathbf{1} & \textit{(contraposée),} \\
(((x \wedge \neg y) \to \mathbf{0}) \to (x \to y)) \equiv \mathbf{1} & \textit{(réduction à l'absurde),} \\
(x \vee (x \wedge y)) \equiv x & \textit{(absorption).}
\end{array}
$$

Exercice 15 *Montrez que si on ajoute aux équivalences de la table 2.1 l'équivalence $(x \vee y) \equiv (x \oplus y)$ alors on peut dériver par un raisonnement équationnel $\mathbf{0} \equiv \mathbf{1}$.*

2.3 Définissabilité et formes normales

Notation Si ℓ est un littéral, on prétendra parfois que $\neg\ell$ est aussi un littéral. Ceci est justifié par le caractère *involutif* de la négation, à savoir on peut toujours remplacer $\neg\neg x$ par x. Si $\{A_i \mid i \in I\}$ est une famille de formules indexées sur un ensemble fini I on peut écrire :

$$\textstyle\bigwedge\{A_i \mid i \in I\} \quad \text{ou} \quad \bigwedge_{i\in I} A_i, \qquad \text{et} \qquad \bigvee\{A_i \mid i \in I\} \quad \text{ou} \quad \bigvee_{i\in I} A_i\ .$$

Comme la disjonction et la conjonction sont associatives et commutatives, cette notation définit une formule unique à équivalence logique près. Par convention, si I est vide on a :

$$\textstyle\bigwedge \emptyset = \mathbf{1} \qquad \text{et} \qquad \bigvee \emptyset = \mathbf{0}\ . \tag{2.2}$$

Définition 22 (fonction définissable) *Soit A une formule et $x_1, \ldots, x_n$ une liste de variables distinctes telle que $var(A) \subseteq \{x_1, \ldots, x_n\}$. Alors la formule A définit une fonction $f_A : \mathbf{2}^n \to \mathbf{2}$ par :*

$$f_A(b_1, \ldots, b_n) = [\![A]\!][b_1/x_1, \ldots, b_n/x_n] \ .$$

Remarque 5 *Notez que la fonction f_A non seulement dépend de A mais aussi de la liste de variables $x_1, \ldots, x_n$. Par exemple, la formule x définit la première projection par rapport à la liste x, y et la deuxième projection par rapport à la liste y, x.*

Exercice 16 *Montrez que deux formules A et B sont équivalentes ssi elles définissent la même fonction sur une liste des variables dans $var(A) \cup var(B)$.*

Définition 23 (DNF) *On appelle* monôme *une conjonction de littéraux. Une formule est en forme normale disjonctive (DNF pour* Disjunctive Normal Form*) si elle est une disjonction de monômes.*

Définition 24 (CNF) *On appelle* clause *une disjonction de littéraux. Une formule est en forme normale conjonctive (CNF pour* Conjunctive Normal Form*) si elle est une conjonction de clauses.*

Proposition 9 *Toute fonction $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 1$, est définissable par une formule A en forme normale disjonctive (DNF) telle que $var(A) \subseteq \{x_1, \ldots, x_n\}$.*

PREUVE. On construit un tableau de vérité avec 2^n entrées. Si $f(b_1, \ldots, b_n) = 1$ avec $b_i \in \{0, 1\}$ alors on construit un monôme $(\ell_1 \wedge \cdots \wedge \ell_n)$ où $\ell_i = x_i$ si $b_i = 1$ et $\ell_i = \neg x_i$ autrement. La formule A est la disjonction de tous les monômes obtenus de cette façon. Par exemple, si $f(0, 1) = f(1, 0) = 1$ et $f(0, 0) = f(1, 1) = 0$ alors on obtient $A = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$. On note que si f est la fonction constante 0 alors on obtient une disjonction vide qui, par la convention (2.2), est $\mathbf{0}$. □

La formule DNF construite dans la proposition 9 est *unique* si l'on considère que conjonction et disjonction sont associatives et commutatives. Cependant, cette formule n'est pas forcement de taille *minimale* (considérez, par exemple, la fonction constante 1).

Remarque 6 *Les éléments d'un ensemble fini X peuvent être codés par les éléments de l'ensemble $\mathbf{2}^n$ pour n suffisamment grand (certains éléments peuvent avoir plusieurs codes). Toute fonction $f : \mathbf{2}^n \to \mathbf{2}^m$ se décompose en m fonctions $f_1 : \mathbf{2}^n \to \mathbf{2}, \ldots, f_m : \mathbf{2}^n \to \mathbf{2}$. Ainsi toute fonction $f : X \to Y$ où X et Y sont finis peut être définie, modulo codage, par un vecteur de formules du calcul propositionnel. Avec un peu de réflexion, tout objet fini peut être représenté par des formules du calcul propositionnel. Cette puissance de représentation explique en partie la grande variété d'applications possibles du calcul propositionnel.*

Tout ce qui a été dit de la forme DNF peut être transféré à la forme CNF 'par dualité'.

Corollaire 1 *Toute fonction $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 1$, est définissable par une formule A en forme normale conjonctive (CNF) telle que $var(A) \subseteq \{x_1, \ldots, x_n\}$.*

PREUVE. Par la proposition 9 on peut construire une formule A en forme normale disjonctive pour la fonction $NOT \circ f : \mathbf{2}^n \to \mathbf{2}$. Donc la formule $\neg A$ définit la fonction f. On applique maintenant les lois de De Morgan et on obtient :

$$\neg \bigvee_{i\in I}(\bigwedge_{j\in J_i} \ell_{i,j}) \equiv \bigwedge_{i\in I}(\neg(\bigwedge_{j\in J_i} \ell_{i,j})) \equiv \bigwedge_{i\in I}(\bigvee_{j\in J_i}(\neg\ell_{i,j})\ ) \equiv \bigwedge_{i\in I}(\bigvee_{j\in J_i} \ell'_{i,j}\ )\ ,$$

où $\ell'_{i,j} = \neg x_{i,j}$ si $\ell_{i,j} = x_{i,j}$ et $\ell'_{i,j} = x_{i,j}$ si $\ell_{i,j} = \neg x_{i,j}$. Bien sûr, on utilise ici l'équivalence logique $x \equiv \neg\neg x$. □

Exercice 17 *Proposez et justifiez une 'règle' pour construire une CNF à partir de la table de vérité d'une fonction booléenne.*

Exercice 18 *Montrez que la satisfaisabilité d'une formule en DNF et la validité d'une formule en CNF peuvent être décidées en temps linéaire dans la taille de la formule.*

Exercice 19 *(1) Soit* $pair(x_1, \ldots, x_n) = (\Sigma_{i=1,\ldots,n} x_i)$ *mod* 2 *la fonction qui calcule la parité d'un vecteur de bits. Montrez que la formule DNF définissant cette fonction dérivée de la proposition 9 a une taille (exemple 5) exponentielle en* n*. (2) Est-ce possible de définir toutes les fonctions de type* $\mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, avec des formules dont la taille est au plus* n^3 *?*

Exercice 20 *Montrez que toute formule est logiquement équivalente à une formule composée de négations et de conjonctions (ou de négations et de disjonctions).*

Exercice 21 *On se focalise sur l'opérateur conditionnel (définition 16). Montrez que toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, est définissable per une formule qui utilise l'opérateur conditionnel et les formules* $\mathbf{0}$ *et* $\mathbf{1}$*.*

Exercice 22 *On considère l'*ou exclusif*, dénoté par le symbole* $\oplus$ *(définition 16). Montrez que : (1)* $\oplus$ *est associatif et commutatif, (2)* $x \oplus \mathbf{0} \equiv x$ *et* $x \oplus x \equiv \mathbf{0}$*, (3) toute fonction booléenne* $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, est définissable par une formule qui utilise* $\mathbf{1}$*,* $\wedge$ *et* $\oplus$*.*

Exercice 23 *Les fonctions binaires NAND et NOR sont définies par :*

$$NAND(x, y) = NOT(AND(x, y))\ , \quad NOR(x, y) = NOT(OR(x, y))\ .$$

Montrez que toute fonction $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, s'exprime comme composition de la fonction NAND (ou de la fonction NOR). Montrez que les* 4 *fonctions unaires possibles n'ont pas cette propriété et que parmi les* 16 *fonctions binaires possibles il n'y en a pas d'autres qui ont cette propriété.*

Exercice 24 *L'ensemble* $\{0, 1\}$ *équipé avec l'addition et la multiplication modulo* 2 *est un* corps *qu'on dénote par* $\mathbf{Z}_2$*. Notez que la multiplication modulo* 2 *coïncide avec la conjonction mais que l'addition modulo* 2 *coïncide avec le ou exclusif. Chaque polynôme en* n *variables à coefficients dans* $\mathbf{Z}_2$ *définit une fonction* $f : \mathbf{2}^n \to \mathbf{2}$*. Un polynôme* multilinéaire *est un polynôme dans lequel chaque variable peut apparaître avec degré au plus* 1*. Montrez que : (1) Chaque polynôme dans* $\mathbf{Z}_2$ *est équivalent à un polynôme multilinéaire. (2) Toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$ *est définissable par un polynôme multilinéaire sur* $\mathbf{Z}_2$ *en* n *variables.*

Chapitre 3

Systèmes de preuve

3.1 Correction et complétude

Pour l'instant on a considéré un *langage logique* (la logique propositionnelle classique) et une notion de *validité*. Comment s'assurer qu'une formule est valide ? Dans le cas de la logique propositionnelle, on peut envisager de vérifier toutes les affectations mais cette méthode demande 2^n vérifications pour une formule qui contient n variables. De plus pour vérifier la validité de formules en *logique du premier ordre* on aurait à considérer une infinité de cas car les domaines d'interprétation sont infinis.

Une autre possibilité serait d'utiliser les équations de la table 2.1. Pour monter que A est valide, il suffit de dériver $A \equiv \mathbf{1}$ par un raisonnement *équationnel*. Hélas, ce type de raisonnement n'est pas si facile à manier (voir, par exemple, l'exercice 14).

L'objectif est donc de se donner des *axiomes* et des *règles* pour déduire avec un effort fini (et on espère raisonnable) de calcul des formules valides. Par exemple, on pourrait avoir les axiomes $(A1-3)$ et la règle (R) :

$$(A1)\quad \frac{}{A \to (B \to A)} \qquad (A2)\quad \frac{}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}$$

$$(A3)\quad \frac{}{(\neg B \to \neg A) \to (A \to B)} \qquad (R)\quad \frac{A \quad A \to B}{B}\ .$$

À partir des axiomes et des règles on peut construire des *preuves*. Une preuve est un arbre dont les feuilles sont étiquetées par des axiomes et dont les noeuds internes sont étiquetés par des règles d'inférence. La formule qui se trouve à la racine de l'arbre est la formule que l'on démontre et l'arbre (c'est-à-dire la preuve) est un *certificat* de sa validité. Par exemple, en prenant $B = (A \to A)$ et $C = A$ on peut construire une preuve de $A \to A$ par application des axiomes $(A1-2)$ et de la règle (R) (2 fois). On remarquera qu'axiomes et règles sont toujours donnés en forme *schématique*. Par exemple, dans l'axiome $(A1)$ il est entendu qu'on peut remplacer les formules A et B par des formules arbitraires.

Définition 25 (correction et complétude) *On dit qu'un système de preuve est* correct *s'il permet de déduire seulement des formules valides et qu'il est* complet *si toute formule valide peut être déduite.*

Il est trivial de construire des systèmes corrects *ou* complets mais il est beaucoup plus délicat de construire des systèmes corrects *et* complets. On va examiner un système correct et complet proposé par Gerhard Gentzen [Gen34, Gen35]. [1] Une idée générale est d'écrire des règles d'inférence qui permettent de réduire la 'complexité structurale (ou logique)' des formules jusqu'à une situation qui peut être traitée directement par un axiome.

Exercice 25 *Soit $A = \ell_1 \vee \cdots \vee \ell_n$ une clause. Montrez que A est valide si et seulement si une variable propositionnelle x et sa négation $\neg x$ sont présentes dans A.*

Ceci suggère un axiome :

$$\frac{}{x \vee \neg x \vee B} \ ,$$

ou plus en général :

$$\frac{}{A \vee \neg A \vee B} \ .$$

On considère maintenant la situation pour la conjonction et la disjonction.

Exercice 26 *Montrez que si $\models A$ et $\models B$ alors $\models A \wedge B$.*

Ceci suggère une règle pour la conjonction :

$$\frac{A \qquad B}{A \wedge B} \ .$$

Exercice 27 *Montrez que si ou bien $\models A$ ou bien $\models B$ alors $\models A \vee B$.*

Ceci suggère deux règles pour la disjonction :

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B} \ .$$

Comment traiter la négation ? L'exercice suivant montre comment réduire la négation en faisant passer la formule à droite ou à gauche d'une implication.

Exercice 28 *Montrez que :*

$$\begin{array}{lll} \models B \to (\neg A \vee C) & \textit{ssi} & \models (B \wedge A) \to C \ , \\ \models (B \wedge \neg A) \to C & \textit{ssi} & \models B \to (A \vee C) \ . \end{array}$$

Ce type de considérations nous mènent à la notion de *séquent*.

Définition 26 (séquent) *Un séquent est un couple (Γ, Δ) qu'on écrit $\Gamma \vdash \Delta$ d'ensembles finis (éventuellement vides) de formules. Un séquent $\Gamma \vdash \Delta$ est valide si la formule suivante est valide :*

$$(\bigwedge_{A \in \Gamma} A) \to (\bigvee_{B \in \Delta} B) \ .$$

1. Le système composé des axiomes $(A1 - 3)$ et de la règle (R) ci-dessus est aussi correct et complet et il est connu comme système de Hilbert. Sa preuve de complétude est plutôt technique.

$$(Ax)\quad \frac{}{A,\Gamma \vdash A,\Delta}$$

$$(\wedge\vdash)\quad \frac{A,B,\Gamma\vdash\Delta}{A\wedge B,\Gamma\vdash\Delta} \qquad (\vdash\wedge)\quad \frac{\Gamma\vdash A,\Delta \quad \Gamma\vdash B,\Delta}{\Gamma\vdash A\wedge B,\Delta}$$

$$(\vee\vdash)\quad \frac{A,\Gamma\vdash\Delta \quad B,\Gamma\vdash\Delta}{A\vee B,\Gamma\vdash\Delta} \qquad (\vdash\vee)\quad \frac{\Gamma\vdash A,B,\Delta}{\Gamma\vdash A\vee B,\Delta}$$

$$(\neg\vdash)\quad \frac{\Gamma\vdash A,\Delta}{\neg A,\Gamma\vdash\Delta} \qquad (\vdash\neg)\quad \frac{A,\Gamma\vdash\Delta}{\Gamma\vdash\neg A,\Delta}$$

TABLE 3.1 – Calcul des séquents de Gentzen

Par convention, on écrit un séquent $\{A_1,\ldots,A_n\}\vdash\{B_1,\ldots,B_m\}$ comme $A_1,\ldots,A_n\vdash B_1,\ldots,B_m$ et un ensemble $\Gamma\cup\{A\}$ comme Γ,A. D'après la définition 26, on peut voir le symbole $\vdash$ comme une implication et interpréter la virgule comme une *conjonction à gauche* et comme une *disjonction à droite* du symbole $\vdash$.

Dans la table 3.1, on reformule nos idées sur la simplification des formules en utilisant la notion de séquent. Ce système est remarquable par sa simplicité conceptuelle : il comporte un axiome 'identité' qui dit que de A on peut dériver A et des règles d'inférence. Pour chaque opérateur de la logique, on dispose d'une règle qui introduit l'opérateur à gauche du $\vdash$ et d'une autre qui l'introduit à droite.

Exercice 29 *Montrez que : (1) un séquent $A,\Gamma\vdash A,\Delta$ est valide, (2) pour toute règle d'inférence, si les hypothèses sont valides alors la conclusion est valide et (3) pour toute règle d'inférence, si la conclusion est valide alors les hypothèses sont valides.*

Proposition 10 *Le calcul des séquents de Gentzen dérive exactement les séquents valides.*

PREUVE. Par l'exercice 29(1-2), tout séquent dérivable est valide. Donc le système est correct. Soit $\Gamma\vdash\Delta$ un séquent valide. On applique les règles jusqu'à ce que toutes les formules dans les séquents soient des variables. Le lecteur peut vérifier qu'il y a toujours au moins une règle qui s'applique et par l'exercice 29(3), si la conclusion d'une règle est valide alors les séquents en hypothèse de la règle sont aussi valides. Ensuite on remarque qu'un séquent valide dont toutes les formules sont des variables peut être dérivé par application de l'axiome (Ax) ; il s'agit d'une simple reformulation de l'exercice 25. □

Définition 27 (sous-formules) *Soit A une formule. L'ensemble $sf(A)$ des sous formules de A est défini par :*

$$sf(A)=\begin{cases}\{A\} & \text{si } A \text{ variable}\\ \{A\}\cup sf(B) & \text{si } A=\neg B\\ \{A\}\cup sf(B_1)\cup sf(B_2) & \text{si } A=B_1\wedge B_2 \text{ ou } A=B_1\vee B_2\ .\end{cases}$$

Exercice 30 *Explicitez la Σ-algèbre qui correspond à la fonction sous-formule de la définition 27.*

Exercice 31 *Montrez que si un séquent est dérivable alors il y a une preuve du séquent qui contient seulement des sous formules de formules dans le séquent.*

Exercice 32 *Montrez que si le séquent* $\Gamma \vdash \Delta$ *est dérivable alors le séquent* $\Gamma \vdash A, \Delta$ *l'est aussi. On appelle cette règle dérivée* affaiblissement*.*

Exercice 33 *Dans le système de Gentzen on peut donner un traitement direct de l'implication :*

$$(\to\vdash) \quad \frac{\Gamma \vdash A, \Delta \quad B, \Gamma \vdash \Delta}{A \to B, \Gamma \vdash \Delta} \quad (\vdash\to) \quad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}$$

Redémontrez la proposition 10 pour le calcul des séquents étendu avec ces règles.

Exercice 34 *Montrez que les règles pour la disjonction et l'implication sont dérivables des règles pour la conjonction et la négation en utilisant les équivalences :* $A \vee B \equiv \neg(\neg A \wedge \neg B)$ *et* $A \to B \equiv \neg A \vee B$*.*

Exercice 35 *Utilisez les règles pour la négation et l'implication (exercice 33) pour construire une preuve des séquents suivants :*

$$\vdash (\neg\neg A \to A) \qquad \textit{et} \qquad (A \to B), (A \to \neg B) \vdash \neg A \ .$$

Exercice 36 *La règle de* coupure *(en anglais,* cut*) est :*

$$(\textit{coupure}) \quad \frac{A, \Gamma \vdash \Delta \quad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta} \ .$$

Montrez que le calcul des séquents étendu avec cette règle est toujours correct (et complet). [2]

Exercice 37 *On considère les formules suivantes :*

$$A = (x \vee z) \wedge (y \vee w), \qquad B = (\neg x \vee \neg y) \wedge (\neg z \vee \neg w), \qquad C = (\neg x \vee \neg z) \wedge (\neg y \vee \neg w) \ .$$

1. *Considérez le séquent* $A, C \vdash B$*. S'il est valide, construisez une preuve du séquent, autrement donnez une affectation des variables* x, y, z, w *qui montre qu'il ne l'est pas.*
2. *Même problème pour le séquent* $A, B \vdash C$*.*

Exercice 38 *Fixez une représentation des séquents et programmez une fonction qui prend en argument un séquent et applique les règles du calcul des séquents de la table 3.1 (dans un ordre que vous allez préciser) afin de décider si le séquent est valide.*

3.2 Conséquence logique et compacité

Définition 28 (conséquence logique, sémantique) *Soit* T *un ensemble de formules et* A *une formule. On dit que* A *est une* conséquence logique *de* T *et on écrit* $T \models A$ *si pour toute affectation* v*, si* v *satisfait chaque formule dans* T *alors* v *satisfait* A*.*

2. Le résultat principal de Gentzen est un algorithme pour transformer une preuve avec coupure en une preuve sans coupure (qui peut être beaucoup plus longue).

Remarque 7 *Si T est un ensemble* fini *de formules alors $T \models A$ ssi $\models (\bigwedge_{B\in T} B) \to A$. Donc dans ce cas la notion de conséquence logique se réduit à la notion de validité d'une formule.*

On peut aussi introduire une version *syntaxique* de la notion de conséquence logique.

Définition 29 (conséquence logique, syntaxique) *Soit T un ensemble de formules et A une formule. On écrit $T \vdash A$ si on peut trouver n formules $B_1, \ldots, B_n \in T$ et construire une preuve du séquent $B_1, \ldots, B_n \vdash A$.*

A priori il n'est pas clair que la version syntaxique est équivalente à la version sémantique. En particulier, on peut remarquer que si T est infini on sait que toute preuve de A va utiliser seulement une portion *finie* de T. On va démontrer que cette propriété (dite de compacité) est vraie aussi pour la version sémantique de la conséquence logique. [3]

Définition 30 (ensemble satisfaisable) *Un ensemble (éventuellement infini) de formules T est satisfaisable s'il existe une affectation qui satisfait chaque formule dans T.*

Exercice 39 *Montrez que si T est satisfaisable alors chaque sous ensemble fini de T est satisfaisable.*

On va montrer que la réciproque est aussi vraie.

Définition 31 (ensemble finiment satisfaisable) *Un ensemble T de formules est* finiment satisfaisable *si tout sous ensemble fini de T est satisfaisable.*

Définition 32 (ensemble maximal) *Un ensemble T de formules est* maximal *si pour toute formule A, ou bien $A \in T$ ou bien $\neg A \in T$.*

Dans la suite on s'intéresse à un T qui est finiment satisfaisable et maximal ; pour cet ensemble, on ne peut pas avoir une formule A telle que $A, \neg A \in T$.

Exercice 40 *Montrez que :*

(1) *Si S est un ensemble finiment satisfaisable et maximal alors :*

$$\begin{array}{lll} A \in S & \textit{ssi} & \neg A \notin S \ , \\ A \wedge B \in S & \textit{ssi} & A \in S \quad \textit{et} \quad B \in S \ , \\ A \vee B \in S & \textit{ssi} & A \in S \quad \textit{ou} \quad B \in S \ . \end{array}$$

(2) *Soit S un ensemble de formules finiment satisfaisable et maximal. On définit une affectation v_S par :*

$$v_S(x) = \begin{cases} 1 & \textit{si } x \in S \\ 0 & \textit{si } \neg x \in S \ . \end{cases}$$

Pourquoi cette définition est-elle correcte ? C'est-à-dire pourquoi pour toute variable x, on peut dire que soit $x \in S$ soit $\neg x \in S$ (ou exclusif) ?

(3) *Soit S finiment satisfaisable et maximal. Montrez que S est satisfaisable.*

(4) *Soit T un ensemble de formules. Montrez que s'il existe $S \supseteq T$ finiment satisfaisable et maximal alors T est satisfaisable.*

3. La notion de compacité est bien établie en topologie et dans les espaces métriques et le fait que la propriété en question s'appelle compacité n'est pas un hasard. En effet, on peut la dériver d'un théoreme de topologie sur les espaces compacts dû à Tykhonov.

Exercice 41 *Soit T un ensemble de formules finiment satisfaisable et A une formule. Montrez que soit $T \cup \{A\}$ est finiment satisfaisable soit $T \cup \{\neg A\}$ est finiment satisfaisable.*

Proposition 11 *Si un ensemble de formules T est finiment satisfaisable alors il est satisfaisable.*

PREUVE. Soit $\{A_n \mid n \in \mathbf{N}\}$ une énumération de toutes les formules. On définit $T_0 = T$ et

$$\begin{array}{ll} T_{n+1} & = \begin{cases} T_n \cup \{A_n\} & \text{si } T_n \cup \{A_n\} \text{ est finiment satisfaisable} \\ T_n \cup \{\neg A_n\} & \text{autrement.} \end{cases} \\ S & = \bigcup_{n \in \mathbf{N}} T_n \ . \end{array}$$

On démontre que T_n est finiment satisfaisable par récurrence sur n en utilisant l'exercice 41. On en dérive que S est finiment satisfaisable car si $X \subseteq S$ et X est fini alors $\exists n \ X \subseteq T_n$. On vérifie aussi que S est maximal car pour toute formule A il existe n tel que $A = A_n$ et donc $A \in T_{n+1}$ ou $\neg A \in T_{n+1}$. Donc par l'exercice 40, S est satisfaisable et donc T l'est aussi. □

Corollaire 2 *Si $T \models A$ alors il existe $T_0 \in \mathcal{P}_{fin}(T)$ tel que $T_0 \models A$.*

PREUVE. Par contraposée. Supposons que pour tout $T_0 \in \mathcal{P}_{fin}(T)$ on a $T_0 \not\models A$. Alors pour tout $T_0 \in \mathcal{P}_{fin}(T)$ on a que $T_0 \cup \{\neg A\}$ est satisfaisable. Ceci implique que pour tout $T_0 \in \mathcal{P}_{fin}(T \cup \{\neg A\})$ on a que T_0 est satisfaisable. Par la proposition 11, on en déduit que $T \cup \{\neg A\}$ est satisfaisable et donc que $T \not\models A$. □

Corollaire 3 *$T \vdash A$ ssi $T \models A$.*

PREUVE. ($\Rightarrow$) Cette direction suit de la correction du calcul des séquents. ($\Leftarrow$). Supposons $T \models A$. Alors, par le corollaire 2, il existe $T_0 \in \mathcal{P}_{fin}(T)$ tel que $T_0 \models A$. Ceci est équivalent à dire : $\{B \mid B \in T_0\} \models A$. Par complétude (proposition 10), il suit que le séquent $\{B \mid B \in T_0\} \vdash A$ est dérivable. □

Exercice 42 *On étudie une application amusante de la compacité. Soit $G = (N, A)$ un graphe non-dirigé (il peut être infini). On dit que G est k-coloriable s'il existe une fonction $c : N \to \{1, \ldots, k\}$ telle que $c(n) \neq c(n')$ si n et n' sont deux noeuds adjacents.*

1. *Pour $n \in N$ et $i \in \{1, \ldots, k\}$ on introduit une variable $x_{n,i}$. On suppose que $x_{n,i}$ vaut 1 ssi le noeud n a couleur i. Explicitez les ensembles de formules qui expriment les conditions suivantes :*
   — *Chaque noeud a au moins une couleur parmi $\{1, \ldots, k\}$.*
   — *Chaque noeud a au plus une couleur parmi $\{1, \ldots, k\}$*
   — *Deux noeuds adjacents n'ont pas la même couleur.*
   Soit $T_{G,k}$ l'ensemble des formules qui expriment les 3 conditions.
2. *Montrez que $T_{G,k}$ est satisfaisable ssi G est k-coloriable.*
3. *On dit que G est finiment k-coloriable si tout sous-graphe fini de G est k-coloriable. Montrez que si G est finiment k-coloriable alors l'ensemble de formules $T_{G,k}$ est finiment satisfaisable.*
4. *Conclure que si un graphe est finiment k-coloriable alors il est k-coloriable.*

Chapitre 4

Réfutation et principe de résolution

4.1 Transformation efficace en CNF

Toute formule peut être transformée en une formule équivalente (définition 21) où l'opérateur de négation $\neg$ peut apparaître seulement devant une variable. Pour ce faire il suffit d'appliquer les lois de De Morgan et l'involution de la négation (proposition 8, groupe $(G4)$) :

$$\neg\neg A \equiv A \ , \quad \neg(A \wedge B) \equiv \neg A \vee \neg B \ , \quad \neg(A \vee B) \equiv \neg A \wedge \neg B \ .$$

La taille de la formule obtenue est toujours *linéaire* dans la taille de la formule de départ (exercice 10). Ensuite, pour obtenir une CNF on pourrait juste itérer l'application de la règle de distribution, à savoir (modulo commutation) :

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \ . \tag{4.1}$$

Cependant une application aveugle de cette règle peut produire une formule dont la taille est exponentielle dans la taille de la formule de départ.

Exercice 43 *Construisez une formule A et une stratégie d'application de l'équivalence (4.1) qui montre qu'on peut obtenir une formule CNF dont la taille est exponentielle dans la taille de la formule de départ A.*

Par l'exercice 18, si pour toute formule A on avait une méthode efficace pour construire une CNF équivalente alors on aurait aussi une méthode efficace pour décider de la validité de A. Ce problème étant ouvert (voir section 5.3), en suivant Tseitin [Tse70], on prend un chemin alternatif qui permet d'obtenir une formule en CNF dont la taille est toujours *linéaire* dans la taille de la formule de départ et qui est *satisfaisable* ssi la formule de départ est satisfaisable.

Soit A une formule qui n'est pas en CNF. Alors on peut trouver une sous-formule B (définition 27) de A de la forme $(\ell \ op \ \ell')$ où ℓ et ℓ' sont des littéraux et op est soit $\wedge$ soit $\vee$. Ensuite on peut construire une formule A' qui contient une nouvelle variable x exactement une fois avec la propriété que $[B/x]A' = A$.

Proposition 12 *La formule A est satisfaisable ssi la formule suivante (4.2) est satisfaisable :* [1]

$$(A' \wedge (x \leftrightarrow (\ell \ op \ \ell'))) \ . \tag{4.2}$$

1. La formule A n'est pas équivalente à la formule (4.2) ; une formule équivalente est $\exists x \ (A' \wedge (x \leftrightarrow (\ell \ op \ \ell')))$ où $\exists x \ C$ est une abréviation pour $([\mathbf{0}/x]C \vee [\mathbf{1}/x]C)$.

PREUVE. Pour toute affectation v, on vérifie, en utilisant la proposition 7, que $v \models A$ ssi $v[[\![B]\!]v/x] \models A'$. □

Maintenant, il est facile de calculer une formule C en CNF équivalente à $x \leftrightarrow (\ell \ op \ \ell')$. Cette formule a une taille bornée par une (petite) constante. Si A' est en CNF on termine, autrement on itère la méthode sur A' dont la taille est inférieure à celle de la formule A car on a remplacé la formule B par la variable x.

Exercice 44 *Construisez une formule CNF qui est satisfaisable ssi la formule suivante est satisfaisable :* $x_1 \leftrightarrow (x_2 \leftrightarrow (x_3 \leftrightarrow (x_4 \leftrightarrow x_5)))$.

Exercice 45 *Programmez une fonction efficace qui prend une formule A et calcule une formule CNF A' qui est satisfaisable ssi A est satisfaisable.*

4.2 Résolution

La méthode de résolution, popularisée par John Alan Robinson [Rob65], peut être vue comme un système de preuve spécialisé dans lequel on utilise une seule règle d'inférence.

Exercice 46 *Montrez que la règle d'inférence suivante est valide, c'est-à-dire la validité des hypothèses implique la validité de la conclusion.*

$$\frac{A \vee \neg D \qquad B \vee D}{A \vee B} \ . \tag{4.3}$$

On va reformuler la règle 4.3 pour des formules en CNF. D'abord, on introduit une notation *ensembliste* pour représenter ces formules. Cette notation nous permet d'utiliser implicitement les règles d'associativité, commutativité et idempotence de la disjonction et de la conjonction (table 2.1, groupe $(G1)$).

Définition 33 (CNF, notation ensembliste) *Dans la notation ensembliste :*
- *une clause C est un ensemble de littéraux (peut être vide),*
- *une formule CNF A est un ensemble de clauses (peut être vide).*

Définition 34 (substitution, notation ensembliste) *Par rapport à la notation ensembliste, on définit la substitution $[b/x]A$ d'une valeur booléenne $b \in \{0,1\}$ dans la CNF A comme suit (ici $\mathbf{1}$ dénote une clause valide) :*

$$\begin{aligned} [b/x]A &= \{[b/x]C \mid C \in A \ et \ [b/x]C \neq \mathbf{1}\} \ , \\ [b/x]C &= \begin{cases} \mathbf{1} & si \ (b = 1 \ et \ x \in C) \ ou \ (b = 0 \ et \ \neg x \in C) \\ C \backslash \{\ell\} & si \ (b = 1 \ et \ \ell = \neg x \in C) \ ou \ (b = 0 \ et \ \ell = x \in C) \\ C & autrement. \end{cases} \end{aligned}$$

Définition 35 (règle de résolution) *Avec cette notation, on formule une variante de la règle (4.3).*

$$\frac{A \cup \{C \cup \{x\}\} \cup \{C' \cup \{\neg x\}\} \quad x \notin C \quad \neg x \notin C'}{A \cup \{C \cup \{x\}\} \cup \{C' \cup \{\neg x\}\} \cup \{C \cup C'\}} \ . \tag{4.4}$$

On appelle (4.4) règle de résolution.

L'effet de l'application de la règle (4.4) consiste à ajouter une nouvelle clause $C \cup C'$ qu'on appelle *résolvant* des deux clauses $C \cup \{x\}$ et $C' \cup \{\neg x\}$. On remarque au passage que sans les conditions $x \notin C$ et $\neg x \notin C'$ on pourrait par exemple 'simplifier' les clauses $\{x\}$ et $\{\neg x\}$ en $\{x, \neg x\}$.

Exercice 47 *Montrez que l'hypothèse de la règle 4.4 est logiquement équivalente à la conclusion. Conclure que si la conclusion n'est pas satisfaisable alors l'hypothèse n'est pas satisfaisable. En particulier, si la conclusion contient la clause vide alors l'hypothèse n'est pas satisfaisable.*

Exercice 48 *Soit A une formule en CNF avec m variables et n clauses. Donnez une borne supérieure en fonction de m et n au nombre de façons d'appliquer la règle de résolution* (4.4).

On va montrer que si une formule A en CNF n'est pas satisfaisable alors la règle de résolution permet de dériver une formule A' avec une clause vide. On dit que la règle de résolution est *complète pour la réfutation*, c'est-à-dire pour la dérivation de la clause vide. Une *application* typique de la méthode est la suivante. On a une formule A en CNF (par exemple, A pourrait être une *base de connaissances*) et on cherche à savoir si la formule C est une conséquence logique de A, c'est-à-dire si $A \to C$ est valide. On observe que $A \to C \equiv \neg A \vee C$ et donc $\neg(A \to C) \equiv (A \wedge \neg C)$. Il suit que la formule $A \to C$ est valide ssi la formule $(A \wedge \neg C)$ n'est pas satisfaisable. On peut donc calculer une formule CNF C' *équivalente* à la formule $\neg C$ et appliquer la méthode de résolution à la CNF $A \wedge C'$.

4.3 Complétude de la résolution pour la réfutation

Il est facile de vérifier (exercice 25) qu'une clause est valide si et seulement si elle contient une variable et sa négation. On appelle une telle clause *triviale*. Les clauses triviales n'ont aucun impact sur la recherche d'une réfutation. On peut éliminer les clauses triviales au début du calcul et ensuite s'abstenir de générer une clause résolvante triviale. Dans la suite, on traite seulement des clauses non-triviales.

Soit A une CNF avec n variables. Si on itère la règle de résolution sur A on sait à priori qu'on peut obtenir au plus 3^n clauses (non-triviales) différentes. En effet chaque variable peut paraître soit positivement, soit négativement soit ne pas paraître du tout. La terminaison de la méthode est donc assurée et la correction suit de l'exercice 47. Il reste deux questions à régler.

- Il faut s'assurer que si A n'est pas satisfaisable alors la méthode de résolution va bien générer la clause vide (la propriété de complétude de la résolution qu'on a déjà évoquée).
- Pour des raisons d'efficacité, il est souhaitable de contrôler le nombre de clauses à traiter. En d'autres termes, on aimerait éliminer autant que possibles celles qui ne sont plus nécessaires à la génération de la clause vide.

Considérons d'abord deux stratégies qui permettent de réduire la taille de la CNF *et* le nombre de variables.

Définition 36 (variable monotone) *Une variable x paraît positivement (négativement) dans A si tous les littéraux qui contiennent x sont de la forme x ($\neg x$). Une variable x est monotone dans A si elle paraît soit positivement soit négativement (mais pas dans les deux formes).*

Exercice 49 *Soit A une CNF et soit x une variable monotone dans A. Montrez que A est satisfaisable ssi $\{C \in A \mid x \notin var(C)\}$ est satisfaisable.*

Définition 37 (clause unitaire) *Une clause est unitaire si elle contient exactement un littéral.*

Exercice 50 *Soit A une CNF et soit $\{x\} \in A$ ($\{\neg x\} \in A$) Montrez que A est satisfaisable ssi $[1/x]A$ ($[0/x]A$) est satisfaisable.*

Il reste donc à traiter le cas d'une variable x dans A qui est ni monotone ni dans une clause unitaire. Pour ce faire on va introduire un opérateur R_x qui va calculer dans un coup toutes les clauses résolvantes par rapport à x et éliminer en même temps x de la formule.

Remarque 8 *La stratégie est similaire à celle adoptée, par exemple, dans la résolution d'un système d'équations linéaires avec la méthode de Gauss. Dans la méthode de Gauss on a un système d'équations qu'il faut satisfaire alors que dans la méthode de résolution on a un système, ou de façon équivalente une conjonction, de clauses qu'il faut satisfaire. En général, l'élimination d'une variable dans une CNF est une opération beaucoup plus coûteuse que dans un système d'équations linéaires (voir remarque 9). Ceci dit, l'exercice 56 montre que pour certaines formules il est possible d'obtenir un algorithme efficace en s'appuyant sur la méthode de Gauss.*

Définition 38 (résolution d'une variable) *Si A est une CNF et $x \in var(A)$ on définit :*

$$\begin{aligned} R_x(A) \quad &= \{C \mid C \in A \text{ et } x \notin var(C)\} \cup \\ &\quad \{C_0 \cup C_1 \mid C_0 \cup \{x\} \in A, C_1 \cup \{\neg x\} \in A, x \notin var(C_i), i = 0,1, C_0 \cup C_1 \text{ pas triviale.}\} \end{aligned}$$

Exercice 51 *Montrez que si A est satisfaisable et $x \in var(A)$ alors $R_x(A)$ est satisfaisable.*

La réciproque est aussi vraie ! On peut donc réduire la réfutation de A à la réfutation de $R_x(A)$.

Proposition 13 *Si A n'est pas satisfaisable et $x \in var(A)$ alors $R_x(A)$ n'est pas satisfaisable.*

PREUVE. Par contradiction on suppose que $v \models R_x(A)$. Notez que par définition $x \notin var(R_x(A))$. Soient $v_0 = v[0/x]$ et $v_1 = v[1/x]$. On sait que $v_i \not\models A$ pour $i = 0, 1$. Donc on peut trouver deux clauses $C_0, C_1 \in A$ telles que $v_0 \not\models C_0$ et $v_1 \not\models C_1$. Comme $v \models R_x(A)$ on doit avoir $x \in var(C_0) \cap var(C_1)$. En particulier on doit avoir $C_0 = C_0' \cup \{x\}$ car autrement $v_0 \models C_0$ et aussi $C_1 = C_1' \cup \{\neg x\}$. Il en suit que $C_0' \cup C_1' \in R_x(A)$ et donc que $v \models C_0' \cup C_1'$. Si $v \models C_0'$ alors $v_0 \models C_0$; contradiction. Et si $v \models C_1'$ alors $v_1 \models C_1$; contradiction. □

La table 4.1 propose un algorithme correct et complet pour la réfutation d'une CNF A (non-triviale). L'algorithme retourne 1 si la formule est satisfaisable et 0 si elle est réfutable. Pour des raisons d'efficacité, il est entendu que le cas $(i+1)$ s'applique seulement si le cas (i) ne s'applique pas. Aussi on remarquera qu'on pourrait omettre le cas (3), car si x est monotone dans A alors le calcul du cas (3) coïncide avec le calcul du cas (6). [2]

2. La stratégie de résolution décrite dans la table 4.1 a été choisie par sa simplicité mais elle n'est pas forcement la plus efficace. D'ailleurs, l'efficacité de la méthode va aussi dépendre de la structure de données choisie pour représenter une CNF ; l'exercice 54 est une invitation à réfléchir à la question.

$Res(A) =$	analyse de cas
(1) $A = \emptyset$	: 1
(2) $\emptyset \in A$	: 0
(3) x monotone dans A	: $Res(\{C \in A \mid x \notin var(C)\})$
(4) $\{x\} \in A$	: $Res([1/x]A)$
(5) $\{\neg x\} \in A$	: $Res([0/x]A)$
(6) $x \in var(A)$	: $Res(R_x(A))$

TABLE 4.1 – Méthode de résolution

Exemple 8 *On considère les clauses suivantes :*

$$C_1 = \{x, y, \neg z\}, \quad C_2 = \{x, y, w, z\}, \quad C_3 = \{x, \neg y, w\}, \quad C_4 = \{\neg x, \neg z\} \ .$$

On veut savoir si la CNF $A = \{C_1, C_2, C_3, C_4\}$ a comme conséquence logique $\{w\}$. Ceci est équivalent à déterminer si $A_0 = A \cup \{C_5\}$ avec $C_5 = \{\neg w\}$ a une réfutation.

- *Il n'y a pas de variable monotone mais C_5 est une clause unitaire. Par le cas (5), on obtient : $\{x, y, \neg z\}$, $\{x, y, z\}$, $\{x, \neg y\}$, $\{\neg x, \neg z\}$.*
- *Il n'y a pas de variable monotone ou de clause unitaire. Par le cas (6) appliqué à y, on obtient : $\{x, \neg z\}$, $\{x, z\}$, $\{\neg x, \neg z\}$.*
- *On applique le cas (6) à x et on obtient juste $\{\neg z\}$ car la clause $\{z, \neg z\}$ est triviale.*
- *Comme z est monotone, par le cas (3) on reste avec un ensemble de clauses vide et par le cas (1) on peut conclure que A_0 est satisfaisable et donc que w* n'est pas *une conséquence logique de A.*

Remarque 9 *Au pire, le cas (6) produit une croissance quadratique de la formule à traiter. Il y a des situations défavorables (exemple à suivre dans la section 4.4) où on a une suite de croissances quadratiques de la formule ce qui donne à la fin une formule de taille exponentielle.*

Exercice 52 *Même question que dans l'exemple 8 mais avec $C_3 = \{\neg y, w\}$ et $C_4 = \{\neg x\}$.*

Exercice 53 *Soit $G = (N, A)$ un graphe non-dirigé fini. Écrivez une formule du calcul propositionnel en CNF qui est réfutable ssi le graphe G n'est pas 3-coloriable (exercice 42).*

Exercice 54 *Fixez une représentation des clauses et des formules CNF et programmez la méthode de résolution (table 4.1).*

4.4 Problèmes faciles et difficiles

Clauses de Horn

Définition 39 (clause et formule de Horn) *Une clause de Horn est une clause (c'est-à-dire une disjonction de littéraux) qui contient au plus un littéral sans négation (positif). Une formule de Horn est une formule en CNF dont les clauses sont des clauses de Horn.*

Un formule de Horn est équivalente à la conjonction (éventuellement vide) de clauses de Horn de la forme suivante :

$$(1) \quad \emptyset \ , \qquad (2) \quad \{x\} \ , \qquad (3) \quad \{\neg x_1, \ldots, \neg x_n\} \ , \qquad (4) \quad \{\neg x_1, \ldots, \neg x_n, x_{n+1}\} \ ,$$

où $n \geq 1$ et $x_i \neq x_j$ si $i \neq j$. Dans ce cas on dit que la formule de Horn est réduite (une telle formule de ne contient pas de clauses triviales).

Proposition 14 *Une formule de Horn réduite qui ne contient pas de clauses de la forme (1) et (2) est satisfaisable.*

PREUVE. Il suffit d'affecter 0 à toutes les variables pour satisfaire les clauses de la forme (3) et (4). □

A cause de la proposition 14, pour les formules de Horn l'algorithme *Res* (table 4.1) peut être grandement simplifié, à savoir si on arrive au dernier cas (6), on retourne directement la valeur 1 (la formule est satisfaisable) sans calculer les résolvants avec l'opérateur R_x ; l'algorithme ainsi simplifié a une complexité polynomiale.

Exemple 9 *Soient* $A = \{C_1, C_2, C_3, C_4, C_5\}$ *avec :* $C_1 = \{\neg x, \neg y, z\}$, $C_2 = \{\neg x, \neg y, \neg w, \neg z\}$, $C_3 = \{\neg x, y, \neg w\}$, $C_4 = \{x, \neg y, \neg w\}$ *et* $C_5 = \{w\}$.

— *On sélectionne* w *et on obtient :* $\{\neg x, \neg y, z\}$, $\{\neg x, \neg y, \neg z\}$, $\{\neg x, y\}$, $\{x, \neg y\}$.

— *Il n'y a pas de variable monotone ou de clause unitaire donc* A *est satisfaisable.*

Exercice 55 *Une formule 2-CNF (3-CNF) est une formule CNF dont toutes les clauses ont au plus* 2 *(*3*) littéraux. Montrez que l'algorithme dans la table 4.1 est efficace (temps polynomial) sur les formules 2-CNF. Expliquez pourquoi votre argument ne s'applique pas aux formules 3-CNF.*

Exercice 56 *Une XCNF est une conjonction de clauses* exclusives *et une clause exclusive est une disjonction exclusive (un xor) de littéraux. (1) Expliquez comment transformer une XCNF en un système d'équations linéaires dans* $\mathbf{Z}_2$ *(voir exercice 24) qui a une solution ssi la formule A est satisfaisable. (2) Adaptez la méthode d'élimination de Gauss à la solution d'un système dans* $\mathbf{Z}_2$*. (3) Conclure que la satisfaisabilité d'une formule XCNF est décidable en temps polynomial.*

Problème des tiroirs

Il est possible de construire des formules pour lesquelles la méthode de résolution va prendre un temps exponentiel dans la taille de la formule. On présente un exemple d'une telle formule sans entrer dans les détails de la preuve. On dispose de m chaussettes et n tiroirs. Le problème des tiroirs $T(m, n)$ a une solution si : (i) chaque chaussette a un tiroir et (ii) chaque tiroir contient au plus une chaussette. En d'autres termes, il faut que la fonction des chaussettes aux tiroirs soit *injective*, ce qui est impossible si $m > n$ (il y a plus de chaussettes que de tiroirs).[3] Il se trouve que toute réfutation de l'assertion $T(n+1, n)$ en utilisant le principe de résolution a une longueur exponentielle en n [Hak85]. On formule le problème $T(m, n)$ en CNF en introduisant les variables $x_{i,j}$ pour $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$. On interprète $x_{i,j}$ par *la chaussette* i *est dans le tiroir* j. On a les conditions :

$$
\begin{array}{ll}
(1) & \bigwedge_{i=1,\ldots,m} (\bigvee_{j=1,\ldots,n} x_{i,j}) \ , \\
(2) & \bigwedge_{i=1,\ldots,n,\ j,k=1,\ldots,m, j<k} (\neg x_{j,i} \vee \neg x_{k,i}) \ , \\
(3) & \bigwedge_{i=1,\ldots,m,\ j,k=1,\ldots,n, j<k} (\neg x_{i,j} \vee \neg x_{i,k}) \ .
\end{array}
$$

3. En anglais, le même problème est formulé en remplaçant chaussettes et tiroirs par pigeons et nids, respectivement. Le fait (trivial) que si $m > n$ alors il y a un tiroir avec au moins deux chaussettes est connu comme *pigeon principle*.

La condition (1) dit que chaque chaussette doit se trouver dans un tiroir, la (2) que chaque tiroir ne peut pas contenir deux chaussettes et la (3) que chaque chaussette ne peut pas être dans deux tiroirs. [4] On prend la conjonction des trois CNF ci dessus, qui est encore une CNF dont la taille est $O(m^2n + mn^2)$.

Exercice 57 *Construisez la formule A en CNF pour le problème $T(2,1)$ et dérivez la clause vide en utilisant la règle de résolution. Même question pour le problème $T(3,2)$.*

4. Le lecteur qui a fait l'exercice 42 devrait avoir une impression de déjà vu. En effet, on obtient les formules ci-dessus aussi dans le cadre de la coloration d'un graphe *complet* (tous les sommets sont adjacents).

Chapitre 5

Satisfaisabilité et méthode DP

5.1 Méthode DP

La méthode de Davis-Putnam-Logemann-Loveland [DP60, DLL62] (abrégé en méthode DP) permet de décider si une formule en CNF est satisfaisable. La méthode est un exemple d'application du principe de *séparation et évaluation* (en anglais, *branch and bound*) qui est une heuristique standard dans la conception d'algorithmes d'optimisation combinatoire. Ici il s'agit d'explorer toutes les affectations possibles en écartant celles qui sont sans issue.

Comme pour la résolution, on représente :

— une formule A en CNF comme un ensemble (éventuellement vide) de clauses $\{C_1, \ldots, C_n\}$,
— une clause C comme un ensemble (éventuellement vide) de littéraux,
— on utilise la notation $[b/x]A$ pour substituer une valeur booléenne $b \in \{0, 1\}$ pour une variable x dans la CNF A.

Ensuite, on définit dans la table 5.1 une fonction DP qui agit récursivement sur une formule A non-triviale en CNF dans la représentation décrite ci-dessus et décide si la formule est satisfaisable (retourne 1) ou pas (retourne 0). On remarquera que les cas (1–5) sont identiques à ceux discutés pour la résolution (table 4.1). La nouveauté concerne le cas (6). Dans la méthode de résolution, on calcule les résolvants par rapport à une variable x alors que dans la méthode DP on explore les deux possibles affectations de valeurs de vérité à x.

Exercice 58 *Montrez que : (1) si A est une formule en CNF alors $DP(A)$ termine, (2) $DP(A)$ retourne 1 (0) si et seulement si A est satisfaisable (ne l'est pas).*

Exercice 59 *Modifiez la fonction DP pour que : (1) si la formule A est satisfaisable, elle retourne une affectation v qui satisfait A et (2) elle retourne le nombre d'affectations sur les*

$DP(A) =$	analyse de cas
(1) $A = \emptyset$:	1
(2) $\emptyset \in A$:	0
(3) x monotone dans A	$DP(\{C \in A \mid x \notin var(C)\})$
(4) $\{x\} \in A$:	$DP([1/x]A)$
(5) $\{\neg x\} \in A$:	$DP([0/x]A)$
(6) $x \in var(A)$:	$OR(DP([0/x]A), DP([1/x]A))$

Table 5.1 – Méthode DP

variables dans $var(A)$ qui satisfont A (si $\sharp var(A) = n$, ce nombre est compris entre 0 et 2^n).

Exercice 60 *Analysez la satisfaisabilité et la validité des formules suivantes :*

$$(x \to w) \to ((y \to z) \to ((x \vee y) \to w)) \ , \qquad (x \to y) \to ((y \to \neg w) \to \neg x) \ .$$

Calculez les CNF des deux formules et de leurs négations. Appliquez la méthode DP pour déterminer la satisfaisabilité des formules obtenues.

Exercice 61 *On considère les formules en CNF suivantes :*

$$\neg x \vee (\neg y \vee x),$$
$$(x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (x \vee \neg y) \wedge \neg x,$$
$$(x \vee y) \wedge (z \vee w) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg w).$$

Pour chaque formule : (i) si la formule est valide, calculez une preuve de la formule dans le calcul des séquents de Gentzen (chapitre 3), (ii) si la formule n'est pas satisfaisable, dérivez la clause vide en utilisant la méthode de résolution (chapitre 4.4), (iii) si la formule est satisfaisable mais pas valide, calculez une affectation qui satisfait la formule en utilisant la méthode DP.

Exercice 62 *Soit A une CNF (en notation ensembliste) telle que $C, C' \in A$ et $C \subset C'$. Montrez que A est satisfaisable ssi $A \backslash \{C'\}$ est satisfaisable.* [1]

Exercice 63 *En utilisant la représentation définie pour l'exercice 54, programmez la méthode DP.*

5.2 Problème 2-SAT

Certains problèmes difficiles pour la résolution comme le principe des tiroirs (section 4.4) le sont aussi pour la méthode DP. Du côte des problème faciles, le cas des formules de Horn (définition 39) est traité exactement comme dans la méthode de résolution : à savoir on utilise la proposition 14 qui affirme qu'une formule de Horn qui ne contient ni une clause vide ni une clause unitaire de la forme $\{x\}$ est toujours satisfaisable en affectant la valeur 0 à toutes les variables.

On pourrait penser qu'une façon de contrôler la difficulté du problème de la satisfaisabilité d'une formule en CNF (abrégé en problème SAT) pourrait être de borner le nombre de littéraux par clause. Il se trouve que le problème où on a 3 littéraux par clause (problème 3-SAT) est aussi difficile que le problème général.

Proposition 15 *Soit A une formule en CNF qui ne contient pas la clause vide. On peut construire en temps linéaire une formule B en CNF avec les propriétés suivantes : (1) A est satisfaisable ssi B est satisfaisable, (2) toutes les clauses dans B ont 3 littéraux et (3) la taille de B est linéaire dans la taille de A.*

1. Cette propriété connue comme *clause subsumption* peut être utilisée pour simplifier les CNF.

PREUVE. Si une clause a la forme $\{\ell\}$ on introduit deux nouvelles variables y_1 et y_2 et on construit les clauses $\{\ell, y_1, y_2\}$, $\{\ell, \neg y_1, y_2\}$, $\{\ell, y_1, \neg y_2\}$ et $\{\ell, \neg y_1, \neg y_2\}$. Si une clause a la forme $\{\ell, \ell'\}$ on introduit une nouvelle variable y et on construit les clauses $\{\ell, \ell', y, \}$ et $\{\ell, \ell', \neg y, \}$. Enfin, si une clause a la forme $\{\ell_1, \ldots, \ell_n\}$ avec $n > 3$, on introduit $n-3$ nouvelles variables $y_1, \ldots, y_{n-3}$ et on construit les clauses : $\{\ell_1, \ell_2, y_1\}$, $\{\neg y_1, \ell_3, y_2\}$,..., $\{\neg y_{n-3}, \ell_{n-1}, \ell_n\}$. □

Par contre, le problème de savoir si une CNF avec au plus 2 littéraux par clause est satisfaisable est 'facile'. Ce problème est connu comme problème 2-SAT. Il peut être attaqué avec la méthode de résolution (exercice 55) et on va voir qu'il peut aussi être réduit à un *problème d'accessibilité* dans un graphe associé à la formule. [2]

Définition 40 (graphe associé à 2-SAT) *Soit A une formule CNF dont les clauses sont non-triviales et contiennent exactement deux littéraux. Soit $var(A) = \{x_1, \ldots, x_n\}$. On définit un graphe dirigé G_A avec noeuds $N = \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$. Dans le graphe G_A on a une arête de ℓ à ℓ' ssi $\{\neg\ell, \ell'\} \in A$, en rappelant qu'il faut voir $\neg\neg x$ comme x.*

Remarque 10 *Dans cette construction si on a une arête de ℓ à ℓ' alors on a aussi une arête symétrique de $\neg\ell'$ à $\neg\ell$ car $\{\neg\ell, \ell'\} = \{\neg\neg\ell', \neg\ell\}$. On remarque aussi qu'on peut lire l'existence d'une arête de ℓ à ℓ' comme une implication logique $\ell \to \ell'$ car $(\ell \to \ell') \equiv (\neg\ell \vee \ell')$.*

Proposition 16 *Soit A une formule CNF dont les clauses contiennent exactement deux littéraux. Alors A n'est pas satisfaisable ssi on peut trouver un noeud (variable) x dans G_A et un chemin qui va de x à $\neg x$ et ensuite revient à x.*

PREUVE. Supposons qu'un tel chemin existe et que A est satisfaisable par une affectation v. Disons que $v(x) = 1$. Comme il y a un chemin de x à $\neg x$ on doit avoir au moins une arête de ℓ à ℓ' avec $v(\ell) = 1$ et $v(\ell') = 0$. Mais une telle arête correspond à une clause $\{\neg\ell, \ell'\} \in A$ qui n'est pas satisfaite par v. Contradiction. Si $v(x) = 0$ alors on obtient aussi une contradiction en considérant le chemin de $\neg x$ à x.

D'autre part, supposons que des tels chemins n'existent pas. On peut construire une affectation qui satisfait A en itérant la méthode suivante. On considère un littéral (noeud) ℓ dont l'affectation n'est pas encore fixée et tel qu'il n'y pas de chemin de ℓ à $\neg\ell$. On considère tous les noeuds accessibles de ℓ et on leur affecte la valeur 1. On affecte aussi la valeur 0 aux noeuds qui correspondent aux négations des noeuds accessibles. Ceci est bien défini car s'il y a des chemins de ℓ à x et $\neg x$ alors par la remarque 10 il y aussi des chemins de x et $\neg x$ à $\neg\ell$ et donc un chemin qui va de ℓ à $\neg\ell$. De plus il n'est pas possible que la valeur 0 ait été affectée à un noeud accessible depuis ℓ dans un pas précèdent. Car dans ce cas ℓ est un prédécesseur du noeud en question et il aurait aussi la valeur 0. Cette méthode affecte une valeur de vérité à chaque noeud de façon à satisfaire chaque clause. □

Remarque 11 *Pour vérifier la condition de la proposition 16, il suffit de calculer la clôture transitive de la relation qui définit les arêtes du graphe associé à la formule. Pour un graphe avec n noeuds, ce calcul prend $O(n^3)$ en utilisant un algorithme classique de Warshall [War62] ; et cette borne supérieure peut être améliorée en utilisant des algorithmes qui calculent en temps linéaire les composantes fortement connexes d'un graphe.*

Exercice 64 *Appliquez la proposition 16 au problème $T(3, 2)$ de l'exercice 57.*

2. On peut montrer l'autre direction aussi : le problème d'accessibilité dans un graphe peut être réduit au problème 2-SAT.

5.3 Modélisation

Une grande variété de problèmes combinatoires peuvent être formulés comme problèmes de satisfaisabilité d'une formule CNF (abrégé en SAT).

Le jeu du Sudoku

Le Sudoku est un puzzle en forme de grille. Le but du jeu est de remplir la grille avec des chiffres allant de 1 à 9 en respectant certaines contraintes, quelques chiffres étant déjà disposés dans la grille. La grille de jeu est un carré de neuf cases de côté, subdivisé en autant de carrés identiques, appelés régions. Par exemple :

5	3			7		9		
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6						8	
			4	1	9			5
				8			7	9

La règle du jeu est simple : chaque ligne, colonne et région ne doit contenir qu'une seule fois tous les chiffres de 1 à 9. Formulé autrement, chacun de ces ensembles doit contenir tous les chiffres de 1 à 9.

La recherche d'une solution du problème du Sudoku peut être exprimée comme un problème de satisfaisabilité d'une formule CNF. Pour chaque case (x, y) de la grille (ligne x et colonne y) et chaque valeur $z \in \{1, \ldots, 9\}$ on introduit une variable propositionnelle s_{xyz}.

- Tout d'abord on veut que chaque case (x, y) contienne au moins un chiffre entre 1 et 9. Par exemple, pour la case $(1, 3)$ la clause générée sera : $s_{131} \vee s_{132} \vee s_{133} \vee \cdots \vee s_{139}$.
- Ensuite, chaque chiffre de 1 à 9 apparaît au plus une fois dans chaque ligne. Par exemple, le fait que le chiffre 1 apparaît au plus une fois dans la ligne 1 correspond aux clauses : $(\neg s_{111} \vee \neg s_{121}), (\neg s_{111} \vee \neg s_{131}), \ldots, (\neg s_{111} \vee \neg s_{191}), (\neg s_{121} \vee \neg s_{131}), \ldots$
- De même, chaque chiffre de 1 à 9 apparaît au plus une fois dans chaque colonne.
- Chaque chiffre de 1 à 9 apparaît au plus une fois dans chaque région.
- Enfin, il faut s'occuper des cases pré-remplies dans la grille de départ. Chacune de ces cases correspond à une clause unitaire.

Notez que si on s'arrête là le codage est suffisant, mais on peut ajouter les contraintes suivantes : (a) Chaque case (x, y) contient au plus un chiffre entre 1 et 9. (b) Chaque chiffre apparaît au moins une fois dans chaque ligne. (c) Chaque chiffre apparaît au moins une fois dans chaque colonne. (d) Chaque chiffre apparaît au moins une fois dans chaque région.

Exercice 65 *Écrivez une formule CNF qui est satisfaisable ssi la version réduite aux chiffres* $1, 2, 3, 4$ *du jeu du Sudoku suivant a une solution.*

1			2
4			
	2	4	

Exercice 66 *Pour $n \geq 1$, on introduit n^2 variables propositionnelles $x_{i,j}$ avec $1 \leq i, j \leq n$. Construisez une formule CNF A_n qui a la propriété suivante : une affectation v satisfait A_n ssi il existe une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ telle que $v(x_{i,j}) = 1$ ssi $\pi(i) = j$.*

Exercice 67 *Soit k un nombre naturel. Construisez une formule CNF A_n telle que $var(A_n) = \{x_1, \ldots, x_n\}$ et une affectation v satisfait A_n ssi $\Sigma_{i=1,\ldots,n} v(x_i) \leq k$.*

Codification d'un calcul

Une caractéristique du problème SAT est que la recherche d'une solution est *difficile* car on ne connaît pas d'algorithme polynomial dans la taille de la formule. D'autre part, la vérification d'une solution est *facile*. En effet il suffit d'interpréter la formule par rapport à une affectation qui satisfait la formule ; un calcul qu'on peut effectuer en temps polynomial. On dit que l'affectation en question est un *certificat* de la satisfaisabilité de la formule.

Un autre exemple de problème qui a cette caractéristique est le problème du coloriage d'un graphe fini (exercice 42) : il est difficile de trouver un coloriage mais il est facile de vérifier si on a un coloriage. On peut définir de façon formelle la classe des problèmes qui sont faciles à vérifier (on appelle cette classe NP) et montrer que pour tous ces problèmes il est possible de construire une formule en CNF de taille raisonnable (polynomiale dans la taille du problème initial) qui est satisfaisable ssi le problème initial a une solution. On dit que le problème de la satisfaisabilité est NP-COMPLET.

On note au passage qu'on ne sait pas si les problèmes de la validité ou de la réfutation sont dans la classe NP. L'idée de prendre la preuve comme *certificat* de validité ne marche pas car la preuve peut avoir une taille exponentielle dans la taille de la formule (voir par exemple le problème des tiroirs). Et on ne sait pas non-plus si la classe NP est contenue dans la classe P des problèmes qui peuvent être résolus en temps polynomial ; il s'agit probablement du problème ouvert le plus célèbre en informatique.

Revenons à ce qu'on sait. Un passage clef pour montrer que tout problème dans NP peut être réduit à un problème de satisfaisabilité consiste à montrer que tout calcul qui prend un temps polynomial peut être codifié par une formule de taille polynomiale. Pour simplifier le travail de codage, il convient d'utiliser un modèle de calcul très rudimentaire (la machine de Turing) mais pour lequel on peut montrer que la perte d'efficacité est contrôlée (dégradation polynomial). En particulier, dans ce modèle l'accès aux données n'est pas direct (comme dans un tableau) mais séquentiel (comme dans une liste). Une *configuration* de la machine peut être vue comme une suite de symboles et un *pas de calcul* consiste à modifier les symboles qui se trouvent autour d'un symbole spécial qui représente l'état de la machine. Un *calcul* est une suite de pas de calculs qui mènent d'une configuration initiale à une configuration terminale. Un calcul peut être représenté comme une matrice dont les lignes correspondent aux configurations. A partir de cette codification, il est assez aisé de trouver une description du calcul par une formule propositionnelle. Un peu comme dans l'exemple du Sudoku, on introduit des variables propositionnelles $x_{i,j,k}$ qui valent 1 ssi au pas de calcul i l'élément j contient le symbole k. Ensuite il s'agit d'écrire les contraintes qui expriment les dépendances entre les symboles.

On propose une formalisation de ces idées dans le cadre d'un modèle de calcul encore plus simple que les Machines de Turing, à savoir le modèle des automates finis déterministes dont on rappelle la définition.

Définition 41 (automates finis) *Un* automate fini *(déterministe) (AFD) est un vecteur* $M = (\Sigma, Q, q_o, F, \delta)$ *où* Σ *est un ensemble fini non-vide de symboles,* Q *est une ensemble fini non-vide d'états,* $q_0 \in Q$ *est l'état initial,* $F \subseteq Q$ *est un ensemble (fini) d'états accepteurs et* $\delta : \Sigma \times Q \to Q$ *est une fonction de transition.*

On dénote par Σ^* les *mots finis* sur Σ et par ϵ le *mot vide*. Une *configuration* est un couple $(q, w) \in Q \times \Sigma^*$. La relation binaire $\to$ est définie à partir de la fonction δ et elle permet de passer d'une configuration à une autre. Il s'agit de la plus petite relation (voir section 1.2) qui pour tout $q \in Q$, $a \in \Sigma$ et $w \in \Sigma^*$ satisfait :

$$(q, aw) \to (\delta(a, q), w) \ .$$

On interprète ce pas de calcul en disant que l'automate étant dans l'état q et en lisant le symbole a se place dans l'état $\delta(a, q)$. On dénote par $\overset{*}{\to}$ la plus petite relation réflexive et transitive qui contient $\to$ (exemple 6).

Définition 42 (mot accepté) *On dit que* $M = (\Sigma, Q, q_o, F, \delta)$ *automate fini accepte le mot* $w \in \Sigma^*$ *si* $(q_o, w) \overset{*}{\to} (q, \epsilon)$ *et* $q \in F$.

Pour tout mot w on va construire une formule A_w telle que A_w est satisfaisable ssi w est accepté par M. Supposons $w = a_1 \cdots a_n$. Un calcul à partir de la configuration initiale (q_0, w) a la forme *triangulaire* suivante :

$$\begin{array}{ccccccc} q_o & a_1 & & a_2 & \cdots & a_n \\ & q_1 = \delta(q_o, a_1) & & a_2 & \cdots & a_n \\ & & q_2 = \delta(q_1, a_2) & & \cdots & a_n \\ & & & & \cdots & a_n \\ & & & & & \delta(q_{n-1}, a_n) \ . \end{array}$$

Pour $i = 0, \ldots, n$ et $q \in Q$ on introduit une variable $x_{i,q}$ qui vaut 1 ssi l'état au pas i est q. Pour $i = 0, \ldots, n-1$, $j = i+1, \ldots, n$ et $a \in \{a_1, \ldots, a_n\}$ on introduit une variable $y_{i,j,a}$ qui vaut 1 ssi au pas i le symbole en position j est a. La condition initiale est exprimée par :

$$x_{0,q_0} \wedge y_{0,1,a_1} \wedge y_{0,2,a_2} \wedge \cdots \wedge y_{0,n,a_n} \ .$$

La condition d'acceptation est exprimée par :

$$\bigvee_{q \in F} x_{n,q} \ .$$

La transition $\delta(q, a)$ est traduite par les conditions :

$$x_{i,q} \wedge y_{i,i+1,a} \to x_{i+1,\delta(q,a)} \ ,$$

pour $i = 0, \ldots, n-1$. Il faut ensuite ajouter des conditions pour exprimer le fait qu'au pas i on recopie tous les symboles sauf celui lu par l'automate :

$$y_{i,j,a} \to y_{i+1,j,a} \ ,$$

pour $i = 0, \ldots, n-1$ et $j = i+2, \ldots, n$. Il faut aussi préciser qu'à chaque pas l'automate peut être dans un seul état. A savoir pour $i = 0, \ldots, n$, $q, q' \in Q$ et $q \neq q'$:

$$(\neg x_{i,q} \vee \neg x_{i,q'}) \ .$$

De même, il faut assumer qu'on a au plus un symbole dans chaque cellule. A savoir pour $i = 0, \ldots, n-1$, $j = i+1, \ldots, n$, $a, b \in \Sigma$, $a \neq b$:

$$(\neg y_{i,j,a} \vee \neg y_{i,j,b}) \ .$$

Exercice 68 *Soit $M = (\{a, b\}, \{q_o, q_1\}, q_o, \{q_1\}, \delta)$ un automate fini avec une fonction δ telle que :*

$$\delta(a, q_o) = q_1, \quad \delta(b, q_o) = q_o, \quad \delta(a, q_1) = q_1, \quad \delta(b, q_1) = q_o \ .$$

Pour tout mot $w \in \{a, b\}^*$, écrivez une formule CNF A_w qui est satisfaisable ssi le mot w est accepté par l'automate.

Chapitre 6

Équivalence logique et diagrammes de décision binaire

6.1 Arbres de décision binaire

On sait (exercice 21) que toute fonction booléenne s'exprime par combinaison de l'opérateur ternaire conditionnel et les formules **0** et **1**. Ceci implique que toute formule A est équivalente à une formule B qui utilise l'opérateur conditionnel et les constantes **0** et **1**. Il est intéressante de remarquer que la formule B peut être obtenue par un processus de transformation de la formule A.

Exercice 69 *Vérifiez l'équivalence logique :*

$$A \equiv x \rightsquigarrow [\mathbf{1}/x]A, [\mathbf{0}/x]A \ . \tag{6.1}$$

L'équivalence logique (6.1) est connue comme *expansion de Shannon*. Si $var(A) = \{x_1, \ldots, x_n\}$ on peut appliquer l'expansion à la variable x_1 pour obtenir :

$$x_1 \rightsquigarrow [\mathbf{1}/x_1]A, [\mathbf{0}/x_1]A \ .$$

Maintenant, la variable x_1 n'apparaît plus dans les formules $[\mathbf{1}/x_1]A$ et $[\mathbf{0}/x_1]A$ et on peut leur appliquer l'expansion par rapport à la variable x_2. En continuant de la sorte, on arrive à une formule B qu'on peut visualiser comme un arbre binaire complet [1] qui contient $1 = 2^0$ noeud étiqueté par x_1, 2^1 noeuds étiquetés par x_2,..., 2^{n-1} noeuds étiquetés par x_n et 2^n noeuds étiquetés par **1** ou **0**.

Cette représentation est aussi mauvaise que les tables de vérité (exponentielle dans le nombre de variables) mais elle permet d'expliciter certaines redondances et de viser une représentation plus compacte. En pratique, il s'agit de détecter des sous-arbres communs et de les partager. Pour effectuer l'opération de partage il est nécessaire de travailler avec des structures plus générales que les arbres, à savoir avec des graphes dirigés acycliques (en anglais, DAG pour *directed acyclic graph*). Une propriété importante de cette représentation est qu'étant donné un ordre sur les variables, le diagramme de décision binaire (abrégé en BDD pour *binary decision diagram*) peut être réduit efficacement à une *forme canonique* (une forme normale unique). On parle alors de BDD *ordonné* et *réduit*. En pratique, la représentation canonique est considérablement plus compacte que la représentation comme arbre obtenue par itération de l'expansion de Shannon.

1. Un arbre binaire est complet s'il n'y a pas un arbre binaire de la même hauteur qui a plus de noeuds.

6.2 Diagrammes de décision binaire (BDD)

Règles de construction

Soit $V = \{x_1, \ldots, x_m\}$ un ensemble fini de variables avec un ordre total $x_1 < x_2 < \ldots < x_m$. Par convention, on suppose aussi que $x_m < 0$ et $x_m < 1$. Un diagramme de décision binaire (BDD) par rapport à cet ordre est un *graphe dirigé* avec un ensemble fini de noeuds N et avec les propriétés suivantes :

- Un noeud $n \in N$ est désigné comme noeud racine et tout noeud est accessible depuis la racine.
- Chaque noeud n a une étiquette $v(n) \in V \cup \{0, 1\}$.
- Si $v(n) \in V$ alors les noeud n a deux arêtes sortantes vers les noeuds $b(n)$ et $h(n)$. Par convention, le noeud $b(n)$ (b pour bas) correspond à la branche 0 et le noeud $h(n)$ (h pour haut) à la branche 1. [2]
- si $v(n) \in \{0, 1\}$ alors le noeud n n'a pas d'arête sortante.
- il y a au plus un noeud étiqueté par 0 et un noeud étiqueté par 1.
- Pour tout noeud n, si $v(n) \in V$ alors $v(n) < v(b(n))$ et $v(n) < v(h(n))$. Cette condition force l'acyclicité du graphe. [3]

Sémantique

Chaque BDD β défini selon les règles décrites ci-dessus induit une fonction unique $f_\beta : \mathbf{2}^m \to \mathbf{2}$. Pour calculer la sortie de $f_\beta(c_1, \ldots, c_m)$, $c_i \in \mathbf{2}$ pour $i = 1, \ldots, m$, on se place au noeud racine de β et on progresse dans le graphe jusqu'à arriver à un noeud étiqueté par 0 ou 1. La valeur de l'étiquette est alors la sortie de la fonction. La règle de progression est que si on se trouve dans le noeud n et $v(n) = x_i$ alors on se déplace vers $b(n)$ si $c_i = 0$ et vers $h(n)$ si $c_i = 1$. La syntaxe des BDD garantit que ce chemin existe et est unique et que donc la fonction f_β est bien définie.

Remarque 12 *(1) Il y a des notions plus générales de BDD qui ne seront pas considérées dans ce cours. Dans ce cours, un BDD est toujours donné par rapport à un ordre des variables et chaque variable est examinée au plus une fois dans un chemin de la racine à une feuille. Pour cette classe de BDD, le problème de l'équivalence logique peut être résolu en temps polynomial.*
(2) *Les BDD peuvent aussi être vus comme des* circuits combinatoires *composés de* multiplexeurs *et de constantes* 0 *et* 1. *Une multiplexeur est un circuit avec* 3 *entrées et une sortie qui réalise l'opérateur conditionnel (définition 16). Chaque noeud étiqueté par une variable x correspond à un multiplexeur et les noeuds étiquetés avec* 0 *et* 1 *correspondent à des entrées constantes* 0 *et* 1. *On remarquera que dans le circuit le calcul procède des feuilles vers la racine plutôt que de la racine vers les feuilles.*

Simplification

A partir d'un BDD on applique 2 règles de simplification :

2. En anglais, le noeud bas est souvent dénoté par $l(n)$ (l pour *low*).
3. Dans la représentation graphique d'un BDD, on utilise parfois une ligne pointillée pour l'arête de n à $b(n)$ et une ligne continue pour celle de n à $h(n)$. De façon équivalente, on peut mettre les étiquettes 0 et 1, respectivement, sur les arêtes.

(S1) Soit n un noeud tel que $v(n) \in V$ et $b(n) = h(n) = n'$. Alors toutes les arêtes vers n peuvent être rédirigées sur n', et n peut être éliminé. Cette règle implémente au niveau BDD l'équivalence logique : $x \rightsquigarrow A, A \equiv A$.

(S2) Soient n et n' deux noeuds distincts tels que $v(n) = v(n') \in V$, $b(n) = b(n')$ et $h(n) = h(n')$. Alors toutes les arêtes vers n' peuvent être rédirigées sur n et n' peut être éliminé. Cette règle est spécifique aux graphes car elle permet de *partager* deux occurrences d'une même formule $x \rightsquigarrow A, B$.

On écrit $\beta \mapsto \beta'$ si un BDD β est transformé en β' par une des règles de simplification $(S1)$ ou $(S2)$. On dit qu'un BDD est réduit si aucune règle de simplification s'applique.

Proposition 17 (1) *Si β est un BDD bien formé par rapport à un ordre donné des variables et $\beta \mapsto \beta'$ alors β' est un BDD bien formé par rapport au même ordre.*

(2) *Toute séquence de simplification termine.*

(3) *Si $\beta \mapsto \beta'$ et $\beta \mapsto \beta''$ alors ou bien $\beta' = \beta''$ ou bien il existe β'_1 et β''_1 tels que $\beta' \mapsto \beta'_1$, $\beta'' \mapsto \beta''_1$ et β'_1 et β''_1 sont égaux à renommage des noeuds près.*

(4) *Tout BDD peut être simplifié en une forme normale et cette forme est unique à renommage des noeuds près.*

Preuve. (1) On vérifie que la syntaxe des BDD est respectée. (2) Chaque simplification élimine un noeud. (3) On examine comment les règles de simplification peuvent agir sur la même portion du graphe. (4) Par (2) on arrive à des formes normales et en itérant (3) on peut conclure que ces formes normales sont identiques à renommage des noeuds près. □

Exercice 70 *Calculez le BDD réduit qui définit la fonction $f : \mathbf{2}^3 \to \mathbf{2}$ avec ordre $x < y < z$.*

xyz	000	001	010	011	100	101	110	111
$f(x,y,z)$	0	0	0	1	0	1	0	1

Exercice 71 *Soit $p : \mathbf{2}^n \to \mathbf{2}$ la fonction pour le contrôle de parité, c'est-à-dire*

$$p(x_1, \ldots, x_n) = (\Sigma_{i=1,\ldots,n} x_i) \ mod \ 2 \ .$$

Donnez le schéma et précisez le nombre de noeuds du BDD réduit qui représente la fonction p par rapport à l'ordre $x_1 < \cdots < x_n$.

Exercice 72 *Montrez que dans un BDD* réduit *la satisfaisabilité et la validité peuvent être décidées en temps constant.*

Exercice 73 *Programmez une fonction qui prend en argument un BDD et calcule un BDD équivalent et réduit (il est possible d'effectuer le calcul en traversant le graphe une seule fois).*

Ordre des variables

L'ordre des variables a un effet important sur la taille d'un BDD. Par exemple, considérons la fonction définie par $\bigvee_{i=1,\ldots,n}(x_i \wedge y_i)$. Avec l'ordre $x_1 < y_1 < \cdots < x_n < y_n$ la taille du BDD réduit est linéaire en n alors qu'avec l'ordre $x_1 < \cdots x_n < y_1 < \cdots < y_n$ la taille du BDD réduit est exponentielle en n.

Une bonne heuristique est de garder proche dans l'ordre les variables qui interagissent dans le calcul du résultat. Il est intéressant d'étudier la meilleure et la pire représentation possible pour certaines classes de fonctions.

— Pour les fonctions *symétriques*, c'est-à-dire pour les fonctions dont le résultat est invariant par permutation de l'entrée, la taille du BBD réduit varie entre linéaire et quadratique (voir exercice 74).
— Pour le bit central de la fonction d'addition sur n bits la taille du BDD réduit varie entre linéaire et exponentielle. Le comparateur n bits de l'exercice 76 est un autre exemple de cette situation.
— Pour le bit central de la fonction de multiplication sur n bits la taille du BDD réduit est *toujours* exponentielle [Bry91]. La preuve de ce résultat utilise entre autres une fonction booléenne (aussi connue comme *hidden weighted bit function*) qui est assez simple pour être présentée ici :

$$f(x_1,\ldots,x_n) = x_{s(x_1,\ldots,x_n)} \quad \text{où } s(x_1,\ldots,x_n) = \Sigma_{i=1,\ldots,n} x_i, \quad \text{et } x_0 = 0 \ .$$

Exercice 74 (1) *Montrez qu'une fonction $f : \mathbf{2}^n \to \mathbf{2}$ est symétrique si et seulement si il y a une fonction $h : \{0,\ldots,n\} \to \mathbf{2}$ telle que $f(x_1,\ldots,x_n) = h(\Sigma_{i=1,\ldots,n} x_i)$.*
(2) *Conclure qu'une fonction symétrique est définie par un BDD réduit dont la taille est $O(n^2)$.*

Exercice 75 *On considère la fonction booléenne $f : \mathbf{2}^{2n} \to \mathbf{2}$ telle que*

$$f(x_{n-1},\ldots,x_0,y_{n-1},\ldots,y_0) = 1 \text{ ssi } (x_{n-1}\cdots x_0)_2 \leq (y_{n-1}\cdots y_0)_2 \ ,$$

où $(z_{n-1}\cdots z_0)_2$ est la valeur en base 2 de la suite $z_{n-1}\cdots z_0$. On ordonne les variables de la façon suivante : $x_{n-1} < y_{n-1} < \cdots < x_0 < y_0$. (1) Calculez le BDD réduit qui définit la fonction f pour $n = 3$. (2) Calculez en fonction de n le nombre de noeuds du BDD réduit qui définit la fonction f.

Exercice 76 (1) *Calculez le BDD réduit pour la fonction définie par la formule $(x \wedge y) \vee (w \wedge z)$ avec l'ordre $x < y < w < z$.*
(2) *Calculez le BDD réduit pour un comparateur de 2-bits $\bigwedge_{i=1,2}(x_i \leftrightarrow y_i)$ en utilisant les ordres $x_1 < y_1 < x_2 < y_2$ et $x_1 < x_2 < y_1 < y_2$.*
(3) *Généralisez à un comparateur de n-bits et estimez le nombre de noeuds dans le BDD réduit pour les ordres $x_1 < y_1 < \cdots < x_n < y_n$ et $x_1 < \cdots x_n < y_1 < \cdots < y_n$.*

6.3 Combinaison de diagrammes

Une autre propriété importante des BDD est qu'il est possible d'effectuer les calculs logiques (négation, conjonction, disjonction,...) directement sur les BDD (réduits). Cette possibilité a eu des retombées fort intéressantes dans le domaine de la vérification de circuits [Bry86].

Restriction

Soit β un BDD relativement aux variables $x_1 < \cdots < x_n$. Ce BDD définit une fonction $f_\beta : \mathbf{2}^n \to \mathbf{2}$. Soit $b \in \mathbf{2}$ une valeur de vérité. Pour construire le BDD qui correspond à la fonction :

$$([0/x_i]f_\beta)(x_1,\ldots,x_n) = f_\beta(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n) \ ,$$

il suffit de rédiriger toute arête qui pointe à un noeud n tel que $v(n) = x$ vers $b(n)$. La construction du BDD pour $[1/x_i]f_\beta$ est similaire. On appelle cette opération *restriction*.

$\mathcal{A}(n, n', op)$	= analyse de cas
(1) $v(n) = v(n') \in V$	: let $n'' = new(v(n))$ in
	$b(n'') := \mathcal{A}(b(n), b(n'), op);\ h(n'') := \mathcal{A}(h(n), h(n'), op);\ n''$
(2) $v(n) <_V v(n')$	: let $n'' = new(v(n))$ in
	$b(n'') := \mathcal{A}(b(n), n', op);\ h(n'') := \mathcal{A}(h(n), n', op);\ n''$
(3) $v(n') <_V v(n)$	: let $n'' = new(v(n'))$ in
	$b(n'') := \mathcal{A}(n, b(n'), op);\ h(n'') := \mathcal{A}(n, h(n'), op);\ n''$
(4) $v(n), v(n') \in \mathbf{2}$	: let $b = (v(n)\ op\ v(n'))$ in n_b
(5) $v(n) \in \mathbf{2}$	: let $n'' = new(v(n'))$ in
	$b(n'') := \mathcal{A}(n, b(n'), op);\ h(n'') := \mathcal{A}(n, h(n'), op);\ n''$
(6) $v(n') \in \mathbf{2}$	: let $n'' = new(v(n))$ in
	$b(n'') := \mathcal{A}(b(n), n', op);\ h(n'') := \mathcal{A}(h(n), n', op);\ n''$

TABLE 6.1 – Algorithme pour l'application

Exercice 77 *Montrez que l'application d'une opération de restriction sur un BDD réduit peut ne pas produire un BDD réduit.*

Application

Soient A et B deux formules avec $var(A) \cup var(B) \subseteq \{x_1, \ldots, x_n\}$ et $op : \mathbf{2}^2 \to \mathbf{2}$ une opération binaire. On remarque que l'opération commute avec l'expansion de Shannon (6.1), à savoir :

$$A\ op\ B \equiv x \rightsquigarrow ([\mathbf{1}/x]A\ op\ [\mathbf{1}/x]B), ([\mathbf{0}/x]A\ op\ [\mathbf{0}/x]B)\ . \tag{6.2}$$

On applique cette remarque aux BDD. Soient β, β' deux BDD qui définissent les fonctions $f_\beta : \mathbf{2}^n \to \mathbf{2}$ et $f_{\beta'} : \mathbf{2}^n \to \mathbf{2}$ par rapport aux variables $x_1 < \ldots < x_n$. Dans la table 6.1, on définit un algorithme $\mathcal{A}$ ($\mathcal{A}$ pour *application*) qui construit un BDD $\mathcal{A}(\beta, \beta')$ définissant la fonction :

$$f_\beta(x_1, \ldots, x_n)\ op\ f_{\beta'}(x_1, \ldots, x_n)\ . \tag{6.3}$$

L'algorithme visite les BDD β et β' en profondeur d'abord. En supposant que n et n' soient les racines de β et β', l'appel $\mathcal{A}(n, n', op)$ retourne la racine d'un BDD qui définit la fonction (6.3). Dans la description de l'algorithme on suppose que $new(x)$ est une fonction qui retourne un nouveau noeud n avec $v(n) = x$, que le champs $b(n)$ et $h(n)$ d'un noeud n sont modifiables (la notation ' :=' indique une modification du contenu de ces champs) et que n_b pour $b \in \mathbf{2}$ sont deux noeuds prédéfinis tels que $v(n_b) = b$. Dans le cas (1), les deux noeuds ont la même variable comme étiquette. Dans les cas (2) et (3), un noeud a comme étiquette une variable qui précède dans l'ordre l'étiquette de l'autre noeud qui est aussi une variable. Dans le cas (4), les deux noeuds ont une valeur booléenne comme étiquette. Dans les cas (5) et (6), exactement un noeud a une valeur booléenne comme étiquette.

Remarque 13 *Cet algorithme peut être amené à évaluer plusieurs fois le même couple de sous-diagrammes. Pour éviter cela, on peut utiliser une optimisation standard en* programmation dynamique *qui consiste à garder dans une table de hachage les couples de sous-diagrammes déjà visités. Une deuxième optimisation est d'arrêter les appels récursifs chaque fois qu'on arrive à une feuille d'un des sous-diagrammes avec la propriété que la valeur de la feuille est suffisante pour déterminer le résultat de l'opération op. Enfin, il est possible*

de modifier l'algorithme de façon à ce qu'il recherche à la volée une des simplifications possibles ; de cette façon, on peut générer directement un BDD réduit à partir de BDD réduits. Quand toutes ces optimisations sont mises en oeuvre et étant donné une table d'hachage qui garantit un temps d'accès constant en moyenne, il est possible de montrer que la complexité de l'opération d'application est de l'ordre du produit de la taille des BDD reçus en entrée. En gros, une opération logique peut au plus élever au carré la taille de la représentation ; on appelle cela une propriété de dégradation gracieuse*. Bien sûr, l'itération d'un carré donne un exponentiel ; un phénomène similaire à celui observé dans le cadre de la méthode de résolution.*

Exercice 78 *Utilisez l'algorithme $\mathcal{A}$ pour construire un BDD réduit qui définit la fonction :*

$$AND(NOR(x, y), AND(z, w)) \ .$$

Exercice 79 *Un* additionneur complet *est une fonction $f : \mathbf{2}^3 \to \mathbf{2}^2$ telle que si $f(x, y, z) = (u, w)$ alors u est la somme (modulo 2) de x, y et z et w est la retenue de la somme. (1) Représentez la fonction f par deux BDD réduits. (2) Expliquez comment combiner n additionneurs complets pour construire un additionneur de 2 nombres sur n bits.*

Exercice 80 *Proposez un algorithme qui prend un BDD β par rapport aux variables ordonnées $x_1 < \cdots < x_n$ et calcule le nombre :*

$$\sharp f_\beta^{-1}(1) = \sharp\{(b_1, \ldots, b_n) \in \mathbf{2}^n \mid f_\beta(b_1, \ldots, b_n) = 1\} \ .$$

Exercice 81 *Programmez l'algorithme d'application de la table 6.1.*

Exercice 82 *Soit β un BDD par rapport aux variables ordonnées $x_1 < \cdots < x_n$. Proposez une méthode pour construire un automate fini (définition 41) qui accepte l'ensemble des mots de la forme $b_1 \cdots b_n$ tels que $b_i \in \mathbf{2}$ et $f_\beta(b_1, \ldots, b_n) = 1$.*

Conclusion

En conclusion, on peut se demander quelle méthode il convient de choisir parmi celles discutées dans ce cours (résolution, DP et BDD). La réponse à cette question est complexe et la littérature sur le sujet très vaste. On se limitera à faire quelques remarques élémentaires.

- Les 3 méthodes sont sensibles à l'*ordre des variables*.
- Pour chaque méthode, on peut formuler des problèmes qui prennent un *temps exponentiel* pour n'importe quel ordre des variables.
- Les 3 méthodes visent des *questions* (réfutation, satisfaisabilité, équivalence logique) et des *domaines d'application* (déduction automatique, optimisation combinatoire, vérification de circuits) différents. Par exemple, si on veut comparer résolution et DP il faut plutôt considérer des formules qui ne sont pas satisfaisables (voir, par exemple, [BKPS02]). Dans une autre direction, la comparaison de DP (ou résolution) et BDD devrait plutôt porter sur des problèmes d'équivalence de circuits (voir, par exemple, [US94]).
- Avec la méthode DP on a une *borne linéaire* sur la taille des données qu'il faut traiter ce qui n'est pas le cas pour résolution et BDD (avec ces 2 méthodes on peut assez rapidement épuiser la mémoire).
- Les méthodes par résolution et par BDD procèdent par *transformation de la contrainte* (formule ou BDD) jusqu'à arriver à une forme où la solution est manifeste, par contre la méthode DP procède par une approche de *séparation et évaluation*.
- Les meilleurs algorithmes utilisent un certain nombre d'*heuristiques* (par exemple, dans le choix de la variable à traiter) dont la présentation est omise.
- On trouve de nombreuses méthodes pour la satisfaisabilité qui raffinent la méthode DP mais il est bon de savoir qu'on trouve aussi une classe de méthodes qui s'appuient sur des *méthodes probabilistes* (marches aléatoires) qui ne sont pas du tout abordées dans ce cours (voir, par exemple, [MU05][chapitre 7]).

Bibliographie

[BA12] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer-Verlag, 2012.

[BKPS02] Paul Beame, Richard M. Karp, Toniann Pitassi, and Michael E. Saks. The efficiency of resolution and Davis–Putnam procedures. *SIAM J. Comput.*, 31(4) :1048–1075, 2002.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8) :677–691, 1986.

[Bry91] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with applications to integer multiplication. *IEEE Trans. Computers*, 40 :205–213, 1991.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Commun. ACM*, 5(7) :394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[Gen34] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(2) :176–210, 1934.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39(3) :405–431, 1935.

[GLM97] Jean Goubault-Larrecq and Ian Mackie. *Proof theory and automated deduction*. Kluwer, 1997.

[Hak85] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39 :297–308, 1985.

[Knu12] Donald Knuth. *The art of computer programming. Volume 4A, Combinatorial algorithsms, Part 1*. Addison-Wesley, 2012.

[Knu18] Donald Knuth. The art of computer programming. volume 4B, Combinatorial algorithms, part 2. Version préliminaire, 2018.

[MU05] Michael Mitzenmacher and Eli Upfal. *Probability and computing : randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[Tse70] Grigory Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics, Steklov Mathematical Institute*, pages 115–125, 1970. Traduit du russe.

[US94] Tomás E. Uribe and Mark E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In *Constraints in Computational Logics, First International Conference, CCL'94, Munich, Germany, September 7-9, 1994*, pages 34–49, 1994.

[War62] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1) :11–12, 1962.

Index