Outils Logiques pour l'Informatique

Roberto M. Amadio

Outils Logiques pour l'Informatique*

Roberto M. Amadio
Université Paris 7

18 juillet 2007

Résumé

Ces notes sont une introduction à la logique mathématique et aux techniques de déduction automatique dans le cadre du calcul propositionnel classique avec des applications à la résolution de problèmes combinatoires et à la modélisation et analyse de systèmes informatiques.

1 Introduction

Dans ces notes on aborde les thèmes suivants.

- Calcul propositionnel classique. Interprétation. Formes normales. Méthode de Davis Putnam. Fonctions définissables. Relation avec les circuits combinatoires.
- Système de preuve de Gentzen. Correction, complétude et compacité. Méthode de preuve par résolution.
- Méthodes basées sur les diagrammes de décision binaire. Application à l'analyse de systèmes finis.
- Langages formels et automates finis. Non-déterminisme et déterminisation.
- Notions de calculabilité. Machines de Turing. Énumérations. Décidabilité. Théorème de Rice.
- Notions de complexité. Classes P et NP. Réduction en temps polynomial. Le problème SAT et la notion de NP-complétude. Théorème de Cook-Levin.
- Ordres bien fondés et principe d'induction.
- Méthodes de terminaison. Plongement monotone. Ordres produit et lexicographique. Lemme de König. Ordre sur les multi-ensembles.
- Travail Pratique. Mise en oeuvre d'une procédure de déduction automatique type Davis-Putnam, Résolution,
- Travail Pratique. Utilisation d'un *SAT solver* type SATO ou CHAFF et application à la résolution de problèmes combinatoires type planification, ordonnancement, programmation linéaire sur les entiers,...

On pourra se référer aux textes suivants pour une présentation plus approfondie.

- J. Barwise, *Handbook of mathematical logic* (chapitres rédigés par J. Barwise et H. Schwichtenberg), Elsevier.

*Ces notes sont basées sur un cours que j'ai assuré à l'Université de Paris 7 en 2005 et 2006. Elles sont complétées par une sélection de travaux dirigés et pratiques.

– J. Gallier. *Logic for computer science* (chapitres 1-4), Harper et Row (disponible en ligne).
– J. Goubault-Larrecq et I. Mackie. *Proof theory and automated deduction* (chapitre 1), Kluwer Academic Publishers.
– M. Sipser. *Introduction to the theory of computation* (chapitres 3-7), Thomson.

On trouve aussi plusieurs textes d'introduction à la logique rédigés en français qui comprennent un chapitre sur le calcul propositionnel. Par exemple :

– R. Cori, D. Lascar. *Logique mathématique, tome 1 : calcul propositionnel - cours et exercices*, Dunod.

Le texte

– P. Wolper. *Introduction à la calculabilité*, InterEditions.

comprend une introduction élémentaire aux machines de Turing et à la complexité.

2 Calcul Propositionnel

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.

2.1 Formules

- Soit $V = \{x_1, x_2, \ldots\}$ un ensemble dénombrable de *variables propositionnelles*.
- L'ensemble *Form* des *formules* est le plus petit ensemble tel que $Form \supseteq V$ et si $A, B \in Form$ alors

$$\begin{array}{ll} \neg A & \text{(négation)}, \\ (A \wedge B) & \text{(conjonction)} \\ (A \vee B) & \text{(disjonction)} \end{array}$$

  sont des formules.[1]
- Si $A \in V$ on dit que A est une *formule atomique*.
- Si $A \in V$ ou $A = \neg B$ et $B \in V$ on dit que A est un *littéral*. Dans le premier cas on dit que le littéral est *positif* et dans le deuxième qu'il est *négatif*. On dénote un littéral avec $\ell, \ell', \ldots$
- L'ensemble $Var(A)$ des variables présentes dans la formule A est défini par :

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

2.2 Interprétation

- $\mathbf{2} = \{0, 1\}$ est l'ensemble des *valeurs booléennes*, d'après George Boole. De façon équivalente on peut utiliser $B = \{\mathsf{faux}, \mathsf{vrai}\}$ avec la convention que faux correspond à 0 et vrai à 1.
- Une *affectation* est une fonction *partielle*

$$v : V \rightharpoonup \mathbf{2}$$

  avec domaine de définition $dom(v)$.
- Si v est une affectation, x est une variable propositionnelle et b une valeur booléenne alors $v[b/x]$ est l'affectation définie par

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

- L'interprétation $[\![A]\!]v$ d'une formule A par rapport à l'affectation v est définie par récurrence sur la structure de A en supposant que $Var(A) \subseteq dom(v)$ (autrement l'interprétation n'est pas définie) :

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

[1] Il s'agit d'un exemple de *définition inductive* d'un ensemble dont il sera question dans la section 8.

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

Parfois, il est préférable d'utiliser une notation plus compacte, à savoir : $\overline{x} = NOT(x)$, $x + y = OR(x, y)$ et $x \cdot y = AND(x, y)$.

- On écrit $v \models A$ si $[\![A]\!]v = 1$.
- On dit que A est *satisfiable* s'il existe une affectation v telle que $v \models A$.
- On dit que A est *valide* (ou une *tautologie*) si pour toute affectation v, $v \models A$.

Exercice 2.1 *Montrez que A est valide si et seulement si $\neg A$ n'est pas satisfiable.*

Exercice 2.2 *Si X est un ensemble de variables et v est une affectation alors $v_{|X}$ est la restriction de v à X. Soit A une formule et $X \supseteq Var(A)$. Montrez que si $v_{|X} = v'_{|X}$ alors $[\![A]\!]v = [\![A]\!]v'$. Donc l'interprétation $[\![A]\!]v$ est indépendante des valeurs de l'affectation v sur les variables propositionnelles qui ne sont pas présentes dans A.*

2.3 Substitution

La substitution $[B/x]A$ d'une formule B pour une variable propositionnelle x dans la formule A est définie par :

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

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

Proposition 2.3 $[\![[B/x]A]\!]v = [\![A]\!]v[[\![B]\!]v/x]$.

Idée de la preuve. Par récurrence sur la structure de A. •

2.4 Équivalence logique

On définit les formules :

$$\mathbf{0} =_{def} x \vee \neg x \quad \mathbf{1} =_{def} x \wedge \neg x \quad (A \to B) =_{def} \neg A \vee B \quad (A \leftrightarrow B) =_{def} (A \to B) \wedge (B \to A)$$

Si $\models A \leftrightarrow B$ on dit que A et B sont logiquement équivalentes et on écrit aussi $A \equiv B$.

Exercice 2.4 *Montrez que A et B sont logiquement équivalentes si et seulement si pour toute affectation v, $[\![A]\!]v = [\![B]\!]v$.*

Exercice 2.5 *Montrez :*

$$(A \vee \mathbf{0}) \equiv A, \quad (A \vee \mathbf{1}) \equiv \mathbf{1}, \quad (A \vee B) \equiv (B \vee A),$$
$$((A \vee B) \vee C) \equiv (A \vee (B \vee C)), \quad (A \vee A) \equiv A$$
$$(A \wedge \mathbf{0}) \equiv \mathbf{0}, \quad (A \wedge \mathbf{1}) \equiv A, \quad (A \wedge B) \equiv (B \wedge A),$$
$$((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C)), \quad (A \wedge A) \equiv A,$$
$$(A \wedge B) \vee C \equiv (A \wedge C) \vee (B \wedge C), \quad (A \vee B) \wedge C \equiv (A \wedge C) \vee (A \wedge C),$$
$$\neg\neg A \equiv A, \quad \neg(A \vee B) \equiv ((\neg A) \wedge (\neg B)), \quad \neg(A \wedge B) \equiv ((\neg A) \vee (\neg B)) \ .$$

On appelle les deux dernières équivalences de l'exercice précédent lois de De Morgan. Si $\{A_i \mid i \in I\}$ est une famille de formules indexées sur l'ensemble I on peut écrire :

$$\begin{array}{ccc} \bigwedge\{A_i \mid_{i\in I}\} & \text{ou} & \bigwedge_{i\in I} A_i, \\ \bigvee\{A_i \mid_{i\in I}\} & \text{ou} & \bigvee_{i\in I} A_i \ . \end{array}$$

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 :

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

2.5 Fonctions définissables et formes normales

- Soit $x_1, \ldots, x_n$ une liste de variables distinctes telle que $\{x_1, \ldots, x_n\} \supseteq var(A)$. Une formule A définit une fonction $f_A : \mathbf{2}^n \to \mathbf{2}$ par

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

  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.
- Une formule est en forme normale disjonctive (DNF pour *Disjunctive Normal Form*) si elle est une disjonction de conjonctions de littéraux.
- On appelle *clause* une disjonction de littéraux. Une formule est en forme normale conjonctive (CNF pour *Conjuctive Normal Form*) si elle est une conjonction de clauses.

Théorème 2.6 *Toute fonction $f : \mathbf{2}^n \to \mathbf{2}$ est définissable par une formule A en forme normale disjonctive telle que $Var(A) = \{x_1, \ldots, x_n\}$.*

Idée de la 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 $A = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$. •

Corollaire 2.7 *Toute fonction $f : \mathbf{2}^n \to \mathbf{2}$ est définissable par une formule A en forme normale conjonctive telle que $Var(A) = \{x_1, \ldots, x_n\}$.*

Idée de la preuve. Par le théorème 2.6 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 :

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

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 $A \equiv \neg\neg A$. •

Remarque 2.8 *Tout ensemble fini X peut être codé par les éléments d'un ensemble 2^n pour n suffisamment grand. 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.*

Exercice 2.9 *Montrez que toute formule est logiquement équivalente à une formule composée de négations et de conjonctions (ou de négations et de disjonctions).*

Exercice 2.10 *La taille $|A|$ d'une formule A peut se définir par :*

$$|x| = 1, |\neg A| = 1 + |A|, |A \wedge B| = 1 + |A| + |B|, |A \vee B| = 1 + |A| + |B| \ .$$

Donnez une borne supérieure à la taille d'une formule qui définit une fonction $f : \mathbf{2}^n \to \mathbf{2}$.

Exercice 2.11 (1) *Montrez que :*

$$\bigvee_{i=1,\ldots,m} (\bigwedge_{j=1,\ldots,n_i} \ell_{i,j}) \equiv \bigwedge_{1 \leq i \leq m, 1 \leq k_i \leq n_i} (\ell_{1,k_1} \vee \cdots \vee \ell_{m,k_m})$$

(2) *Supposez $n_i = n$ pour $i = 1, \ldots, m$. Exprimez la taille des formules dans (1) en fonction de n et m.*

(3) *Dérivez une procédure pour transformer une formule en CNF.*

Exercice 2.12 (1) *Montrez l'équivalence logique :*

$$(A \wedge B) \vee (\neg A \wedge B) \equiv B \tag{1}$$

(2) *On peut appliquer cette équivalence logique pour simplifier une forme normale disjonctive. Par exemple, considérez la fonction $f(x, y, z)$ définie par le tableau de vérité :*

$x \backslash yz$	00	01	11	10
0	0	1	1	0
1	1	1	1	1

Calculez la forme normale disjonctive de f et essayez de la simplifier en utilisant l'équivalence logique 1.

(3) *La présentation du tableau de vérité n'est pas arbitraire... Proposez une méthode graphique pour calculer une forme normale disjonctive simplifiée.*

Exercice 2.13 *Soit f une fonction sur les nombres naturels. Dire qu'un problème est décidé en $O(f)$, signifie qu'on dispose d'un algorithme A et de n_0, k nombres naturels tels que pour toute entrée dont la taille n est supérieure à n_0, le temps de calcul de A sur l'entrée en question est inférieure à $k \cdot f(n)$.*

(1) *Montrez que la satisfaction d'une formule en DNF et la validité d'une formule en CNF peuvent être décidées en $O(n)$.*

(2) *Soit $pair(x_1, \ldots, x_n) = (\Sigma_{i=1,\ldots,n} x_i) \bmod 2$ la fonction qui calcule la parité d'un vecteur de bits. Montrez que la représentation en DNF ou CNF de cette fonction est en $O(2^n)$. Peut-on appliquer (1) pour simplifier la représentation ?*

Exercice 2.14 (if-then-else) *La fonction ternaire ITE est définie par* $ITE(1, x, y) = x$ *et* $ITE(0, x, y) = y$. *Montrez que toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 0$ *s'exprime par composition de la fonction ITE et des (fonctions) constantes* 0 *et* 1.

Exercice 2.15 (nand,nor) *Les fonctions binaires NAND et NOR sont définies par* $NAND(x, y) = NOT(AND(x, y))$ *et* $NOR(x, y) = NOT(OR(x, y))$. *Montrez que toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 0$, *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 2.16 *L'*or exclusif $\oplus$ *(xor) est défini par*

$$A \oplus B \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$$

Montrez que :

(1) $\oplus$ *est associatif et commutatif.*

(2) $x \oplus 0 \equiv x$ *et* $x \oplus x \equiv 0$.

(3) *Toute fonction booléenne* $f : \mathbf{2}^n \to \mathbf{2}$ *peut être représentée à partir de* 1, $\wedge$ *et* $\oplus$.

2.6 Méthode de Davis-Putnam

La méthode de Davis Putnam permet de décider si une formule en forme normale conjonctive est satisfiable. On représente une formule A en CNF comme un ensemble (éventuellement vide) de clauses $\{C_1, \ldots, C_n\}$ et une clause C comme un ensemble (éventuellement vide) de littéraux. Dans cette représentation, on définit la substitution $[b/x]A$ d'une valeur booléenne $b \in \{0, 1\}$ dans A comme suit :

$$[b/x]A = \{[b/x]C \mid C \in A \text{ et } [b/x]C \neq 1\}$$

$$[b/x]C = \begin{cases} 1 & \text{si } (b = 1 \text{ et } x \in C) \text{ ou } (b = 0 \text{ et } \neg x \in C) \\ C \backslash \{\ell\} & \text{si } (b = 1 \text{ et } \ell = \neg x \in C) \text{ ou } (b = 0 \text{ et } \ell = x \in C) \\ C & \text{autrement} \end{cases}$$

On définit une fonction DP qui agit récursivement sur une formule A en CNF dans la représentation décrite ci-dessus :

function $DP(A) =$	case
(1) $A = \emptyset$:	true
(2) $\emptyset \in A$	false
(3) $\{x, \neg x\} \subseteq C \in A$:	$DP(A \backslash \{C\})$
(4) $\{x\} \in A$:	$DP([1/x]A)$
(5) $\{\neg x\} \in A$:	$DP([0/x]A)$
(6) *else* :	choisir x dans A; $DP([0/x]A)$ or $DP([1/x]A)$

Dans (1), nous avons une conjonction du vide qui par convention est équivalente à true. Dans (2), A contient une clause vide. La disjonction du vide étant équivalente à false, la formule A est aussi équivalente à false. Dans (3), une clause contient un littéral et sa négation et elle est

donc équivalente à true. Dans (4) et (5), A contient une clause qui est constituée uniquement d'une variable ou de sa négation. Ceci permet de connaître la valeur de la variable dans toute affectation susceptible de satisfaire la formule. Dans (6), nous sommes obligés à considérer les deux valeurs possibles d'une affectation sur une variable.

Exercice 2.17 (1) *Montrez que si A est une fonction en CNF alors la fonction DP termine.*
(2) *Montrez que $DP(A)$ retourne* true *(*false*) si et seulement si A est satisfiable (ne l'est pas).*

Exercice 2.18 *Expliquez comment utiliser la méthode de Davis-Putnam pour décider la validité d'une formule.*

Exercice 2.19 *Modifiez la fonction DP pour que, si la formule A est satisfiable, elle retourne une affectation v qui satisfait A.*

Exercice 2.20 *Réfléchissez aux structures de données et aux opérations nécessaires à la mise en oeuvre de l'algorithme en Java.*

Exercice 2.21 *En logique classique, on peut définir l'implication $A \to B$ comme $\neg A \vee B$. Analysez la satisfiabilité et la validité des formules suivantes :*

$$(x \to w) \to ((y \to z) \to ((x \vee y) \to w))$$
$$(x \to y) \to ((y \to \neg w) \to \neg x)$$

Calculez la CNF des deux formules et de leurs négations. Appliquez la méthode de Davis-Putnam pour déterminer la satisfiabilité des formules obtenues.

Exercice 2.22 *Une clause de Horn est une clause (c'est-à-dire une disjonction de littéraux) qui contient au plus un littéral positif. Une formule de Horn est une formule en CNF dont les clauses sont des clauses de Horn.*
(1) *Montrez que toute formule de Horn est équivalente à la conjonction (éventuellement vide) de clauses de Horn de la forme :*

$$\begin{array}{ll} (1) & x \\ (2) & \neg x_1 \vee \cdots \vee \neg x_n \\ (3) & \neg x_1 \vee \cdots \vee \neg x_n \vee x_{n+1} \end{array}$$

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.
(2) *Montrez qu'une formule de Horn réduite qui ne contient pas de clauses de la forme (1) ou qui ne contient pas de clauses de la forme (2) est satisfiable.*
(3) *Donnez une méthode efficace (temps polynomial) pour déterminer si une formule de Horn est satisfiable.*

2.7 Circuits

Une formule A du calcul propositionnel avec variables $x_1, \ldots, x_n$ peut être vue comme un arbre. On a vu que A définit une fonction $f_A : \mathbf{2}^n \to \mathbf{2}$. Une façon naturelle de calculer la fonction f_A est de propager les valeurs de vérité des feuilles vers la racine. On peut mesurer la complexité du calcul en comptant le nombre de portes ou en comptant la longueur du chemin le plus long. Intuitivement, la première mesure correspond à l'espace occupé par le calcul alors que la deuxième correspond au temps nécessaire au calcul.

2.7.1 Circuits booléens

Une formule/arbre A peut présenter une certaine redondance. Par exemple, considérons la formule :

$$(x_3 \wedge \neg((x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))) \vee (\neg x_3 \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2))$$

Les formules x_1, x_2, x_3, $(x_1 \vee x_2)$, $(\neg x_1 \vee \neg x_2)$ paraissent plusieurs fois dans la formule A. On peut alors envisager de donner une représentation plus compacte de A dans laquelle les sous formules identiques sont partagées. On arrive ainsi à la notion de *circuit booléen*.

Un circuit booléen est un graphe dirigé acyclique $G = (N, A)$ où on appelle les noeuds dans N *portes logiques*. A chaque porte logique on associe une étiquette $\wedge, \vee, \neg, \mathsf{0}, \mathsf{1}, x_1, \ldots, x_n$. Les noeuds avec étiquettes $\wedge, \vee$ ont 2 arêtes entrantes, les noeuds avec étiquette $\neg$ ont 1 arête entrante et les noeuds avec étiquettes $\mathsf{0}, \mathsf{1}, x_1, \ldots, x_n$ n'ont pas d'arête entrante. Les noeuds qui n'ont pas d'arête entrante correspondent aux *entrées* du circuit. Les noeuds qui n'ont pas d'arête sortante correspondent aux *sorties* du circuit. Chaque sortie correspond à une fonction (avec entrées $x_1, \ldots, x_n$) représentée par le circuit.

Les fonctions calculées par le circuit sont obtenues en affectant des valeurs de vérité aux variables $x_1, \ldots, x_n$ et en propageant ces valeurs de vérité des entrées vers les sorties. Le fait que le graphe est acyclique assure que ce calcule peut toujours être effectué et que le résultat est déterminé de façon unique.

2.7.2 Circuits combinatoires

Les circuits booléens sont une abstraction mathématique de dispositifs électroniques qu'on appelle circuits combinatoires. Dans les circuits combinatoires, les portes logiques sont réalisées par des transistors, les arêtes correspondent à des interconnexions et les valeurs booléennes correspondent à des tensions. Typiquement, 0 est représenté par la masse (tension $0V$) et 1 par $5V$.

Dans les circuits combinatoires les boucles sont interdites, ce qui correspond à la condition d'acyclicité dans les circuits booléens. Cette condition permet de garantir que suite à une variation des tensions en entrée, la tension du circuit en sortie se stabilise sur une valeur significative (proche de $0V$ ou de $5V$ après un temps qui est lié à des variables physiques comme la température et la longueur des interconnexions).

Remarquons que les mesures de complexité que nous avons évoquées pour les circuits logiques ont une interprétation immédiate en terme de circuits combinatoires. Le nombre de noeuds du circuit booléen correspond au nombre de portes logiques, c'est-à-dire au nombre de transistors nécessaires à la mise en oeuvre du circuit. Couplée avec la topologie des interconnexions, cette mesure détermine l'espace occupé par le circuit. La longueur du chemin le plus long correspond au temps qu'il faut attendre entre une variation du signal en entrée et la stabilisation du signal en sortie.

La notion de circuit booléen fait abstraction de la notion de temps (le calcul du résultat est instantané) et dans une certaine mesure de distance (on compte le nombre de portes mais on ne compte pas la longueur des interconnexions) et il permet de simplifier grandement la conception d'un circuit combinatoire. Dans la suite nous allons considérer dans un certain détail la conception d'un additionneur.

2.7.3 Additionneur

On considère un vecteur $b_n, \ldots, b_0$ où $b_i \in \{0,1\}$ comme un nombre en base 2. Ainsi le nombre représenté est $\Sigma_{i=0,\ldots,n} b_i 2^i$ qu'on dénote aussi avec $(b_n \cdots b_0)_2$.

Un *multiplexeur* est un circuit booléen avec $n + 2^n$ entrées $c_{n-1}, \ldots, c_0, x_{2^n-1}, \ldots, x_0$ et une sortie y tel que

$$y = x_{(c_{n-1}\cdots c_0)_2}$$

Exercice 2.23 *Construisez un circuit booléen qui réalise un multiplexeur dont le nombre de portes est proportionnel à 2^n et dont la longueur du chemin le plus long est proportionnelle à n.*

Un additionneur est un circuit booléen avec $2n$ entrées $x_{n-1}, y_{n-1}, \ldots, x_0, y_0$ et $n+1$ sorties $r_n, s_{n-1}, \ldots, s_0$ tel que

$$(x_{n-1} \cdots x_0)_2 + (y_{n-1} \cdots y_0)_2 = (r_n s_{n-1} \cdots s_0)_2$$

On peut réaliser un additionneur en utilisant l'algorithme standard qui propage la retenue de droite à gauche.

Exercice 2.24 (1) *Réalisez un circuit A avec 3 entrées x, y, r et deux sorties s, r' tel que*

$$(r's)_2 = (x)_2 + (y)_2 + (r)_2$$

(2) *Expliquez comment inter-connecter n circuits A pour obtenir un additionneur sur n bits.*

(3) *Montrez que dans le circuit en question le nombre de portes et la longueur du chemin le plus long sont proportionnels à n.*

Exercice 2.25 *Le but de cet exercice est de réaliser un additionneur dont le nombre de portes est encore polynomiale en n mais dont la longueur du chemin le plus long est proportionnelle à $lg(n)$. Pour éviter que la retenue se propage à travers tout le circuit, l'idée est d'anticiper sa valeur. Ainsi pour additionner 2 vecteurs de longueur n, on additionne les premiers $n/2$ bits (ceux de poids faible) et en même temps on additionne les derniers $n/2$ bits (ceux de poids fort) deux fois (en parallèle) une fois avec retenue initiale 0 et une fois avec retenue initiale 1. On applique cette méthode récursivement sur les sous-vecteurs de longueur $n/4, n/8, \ldots$ selon le principe* diviser pour régner*.*

(1) *Construisez explicitement un tel circuit pour $n = 4$.*

(2) *Déterminez en fonction de n le nombre de portes et la longueur du chemin le plus long du circuit obtenu.*

Exercice 2.26 *Un décodeur est un circuit avec n entrées $x_{n-1}, \ldots, x_0$ et 2^n sorties $y_{2^n-1}, \ldots, y_0$ tel que*

$$y_i = 1 \text{ ssi } i = (x_{n-1} \cdots x_0)_2$$

Réalisez un tel circuit.

Exercice 2.27 *On dispose d'un circuit combinatoire CE avec 2 entrées x, y et 2 sorties $<, =$ dont le comportement est spécifié par le tableau suivant (bien sûr, les symboles choisis pour les sorties ne sont pas arbitraires) :*

x	y	$<$	$=$
0	0	0	1
0	1	1	0
1	0	0	0
1	1	0	1

Un comparateur n bits est une fonction booléenne C avec $2n$ entrées et 1 sortie telle que :

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

On remarque que :

$$(x_{n-1} \cdots x_0)_2 < (y_{n-1} \cdots y_0)_2 \quad ssi \quad \begin{array}{l} (x_{n-1} < y_{n-1}) \text{ ou} \\ ((x_{n-1} = y_{n-1}) \text{ et } (x_{n-2} \cdots x_0)_2 < (y_{n-2} \cdots y_0)_2) \end{array}$$

Montrez comment construire un circuit combinatoire qui implémente un comparateur 4 bits en disposant de : (i) 4 circuits CE, (ii) 8 portes AND binaires (vous n'êtes pas obligés de les utiliser toutes) et 1 porte OR avec 4 entrées. Si vous êtes bloqué, essayez d'abord le comparateur 2 bits.

3 Systèmes de preuve

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. D'où l'idée de se donner des *axiomes* et des *règles* pour déduire avec un effort fini de calcul des formules valides. Par exemple, on pourrait avoir les axiomes :

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

et on pourrait avoir une règle :

$$(R)\quad \frac{A \quad A \to B}{B}$$

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. 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.

3.1 Correction et complétude

On dit qu'un système de preuve est :

correct s'il permet de déduire seulement des formules valides,

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 en 1930. 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 3.1 *Soit $A = \ell_1 \vee \cdots \vee \ell_n$ une disjonction de littéraux. 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 3.2 *Montrez que :*

$$\models A \wedge B \quad ssi \quad \models A \quad et \quad \models B$$

Ceci suggère une règle pour la conjonction :

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

Exercice 3.3 *Montrez que :*

$$\models A \vee B \quad si \quad \models A \quad ou \quad \models B$$

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 3.4 *Montrez que :*

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

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

Définition 3.5 *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*

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

est valide.

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. On remarquera que la virgule ',' est interprétée comme une *conjonction à gauche* et comme une *disjonction à droite* du séquent. On va maintenant reformuler nos idées sur la simplification des formules en utilisant la notion de séquent.

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

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

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

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

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 3.6 *Montrez que :*

(1) *Un séquent $A, \Gamma \vdash A, \Delta$ est valide.*

(2) *Pour chaque règle d'inférence la conclusion est valide si et seulement si les hypothèses sont valides.*

Théorème 3.7 *Le système de Gentzen dérive exactement les séquents valides.*

IDÉE DE LA PREUVE. Par l'exercice 3.6 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 atomiques. Ensuite on remarque qu'un séquent valide dont toutes les formules sont atomiques peut être dérivé par application de l'axiome (Ax). Cette remarque est une simple reformulation de l'exercice 3.1. •

Définition 3.8 *Soit A une formule. L'ensemble $sf(A)$ des sous formules de A est défini par*

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

Exercice 3.9 (sous-formule) *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 3.10 (affaiblissement) *Montrez que si le séquent $\Gamma \vdash \Delta$ est dérivable alors le séquent $\Gamma \vdash A, \Delta$ l'est aussi.*

Exercice 3.11 (implication) *Dans le système de Gentzen on peut donner un traitement direct de l'implication :*

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

Redémontrez le théorème 3.7 pour le système de Gentzen étendu avec ces règles.

Exercice 3.12 *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 $A \rightarrow B \equiv \neg A \vee B$.*

Exercice 3.13 (1) *Écrire l'axiome et les règles d'inférence du calcul des séquents pour les opérateurs logiques de négation $\neg$ et d'implication $\rightarrow$. Rappel : on peut retrouver les règles pour l'implication à partir des règles pour la négation et la disjonction.*

(2) *Utilisez les système de preuve décrit pour construire une preuve des séquents suivants :*

$$\vdash (\neg\neg A \rightarrow A) \qquad \text{et} \qquad (A \rightarrow B), (A \rightarrow \neg B) \vdash \neg A$$

Exercice 3.14 *Trouvez les règles* $(\vdash NAND)$ *et* $(NAND \vdash)$ *pour l'opérateur logique NAND en utilisant le fait que* $NAND(A, B)$ *s'écrit comme* $\neg(A \wedge B)$.

Exercice 3.15 (coupure) *La règle de* coupure *(ou* cut*) est :*

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

Montrez que le système de Gentzen étendu avec cette règle est toujours correct (et complet).

Exercice 3.16 *Dérivez du système de Gentzen un algorithme pour vérifier si une formule A est valide. Quelle est la complexité en temps de votre algorithme ?*

Exercice 3.17 *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* $A, B \vdash C$.

3.2 Compacité

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

Exercice 3.18 *Si* T *est satisfiable alors chaque sous ensemble fini de* T *est satisfiable.*

On va montrer que la réciproque est aussi vraie.

Définition 3.19 (1) *Un ensemble* T *de formules est* finement satisfiable *si tout sous ensemble fini de* T *est satisfiable.*

(2) *Un ensemble* T *de formules finement satisfiable est* maximal *si pour toute formule* A *soit* $A \in T$ *soit* $\neg A \in T$.

Exercice 3.20 *Montrez que :*

(1) *Si* S *est un ensemble finement satisfiable et maximal alors :*

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

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

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

Pourquoi cette définition est-elle correcte ?

(3) *Soit* S *finement satisfiable et maximal. Montrez que* S *est satisfiable.*

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

Exercice 3.21 *Soit T un ensemble de formules finement satisfiable et A une formule. Alors, soit $T \cup \{A\}$ est finement satisfiable soit $T \cup \{\neg A\}$ est finement satisfiable.*

Théorème 3.22 (compacité) *Si un ensemble de formules T est finement satisfiable alors il est satisfiable.*

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

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

On démontre que T_n est finement satisfiable par récurrence sur n en utilisant l'exercice 3.21. On en dérive que S est finement satisfiable 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 3.20, S est satisfiable et donc T l'est aussi. •

Exercice 3.23 *Soit T un ensemble de formules. On écrit $T \models A$ si pour toute affectation v, si v satisfait T alors v satisfait A. Montrez que si $T \models A$ alors il existe T_0 sous-ensemble fini de T tel que $T_0 \models A$. Suggestion : utilisez le théorème de compacité.*

3.3 Méthode de preuve par résolution

Exercice 3.24 *Montrez que la règle d'inférence suivante est valide :*

$$\frac{A \vee \neg C \qquad B \vee C}{A \vee B} \tag{2}$$

Exercice 3.25 *Pour représenter les formules en CNF on adopte la même notation ensembliste utilisée pour décrire la méthode de Davis-Putnam.*

- *Une clause C est un ensemble de littéraux.*
- *Une formule A est un ensemble de clauses.*

Nous considérons une variante de la règle (2) :

$$\frac{A \cup \{C \cup \{x\}\} \cup \{C' \cup \{\neg x\}\} \quad x \notin C \quad \neg x \notin C'}{A \cup \{C \cup \{x\}\} \cup \{C' \cup \{\neg x\}\} \cup \{C \cup C'\}} \tag{3}$$

Dans la suite on appelle (3) règle de résolution.[2] *L'effet de l'application de la règle consiste à ajouter une nouvelle clause $C \cup C'$ qu'on appelle* résolvant *des deux clauses $C \cup \{x\}$ et $C' \cup \{\neg x\}$.*

(1) *Montrez que l'hypothèse est logiquement équivalente à la conclusion.*

(2) *Conclure que si la conclusion n'est pas satisfiable alors l'hypothèse n'est pas satisfiable. En particulier, si la conclusion contient la clause vide alors l'hypothèse n'est pas satisfiable.*

[2]Sans les conditions $x \notin C$ et $\neg x \notin C'$ on pourrait par exemple 'simplifier' les clauses $\{x\}$ et $\{\neg x\}$ en $\{x, \neg x\}$.

Fait Si une formule A en CNF n'est pas satisfiable 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. La méthode peut être implémentée itérativement. A chaque itération on ajoute toutes les clauses qui sont un résolvant de deux clauses. Cette itération termine forcément car le nombre de clauses qu'on peut construire est fini. Parfois, il convient de représenter la dérivation comme un graphe dirigé acyclique (ou DAG pour *directed acyclic graph*) dont les noeuds sont étiquetés par les clauses. Initialement on a autant de noeuds que de clauses et pas d'arêtes. Chaque fois qu'on applique la règle de résolution (3) on introduit un nouveau noeud qui est étiqueté avec la clause résolvant $C \cup C'$ et deux nouvelles arêtes qui vont des noeuds étiquetés avec les clauses $C \cup \{x\}$ et $C' \cup \{\neg x\}$ vers le noeud étiqueté avec la clause $C \cup C'$.

Exercice 3.26 *Construire la formule A en CNF qui correspond au principe du nid de pigeon avec 2 pigeons et 1 nid. Dérivez la clause vide en utilisant la règle de résolution. Même problème avec 3 pigeons et 2 nids (attention le calcul risque d'être long).*

Exercice 3.27 *Soit A une formule en CNF avec m variables et n clauses. Montrez qu'il y a au plus $m \cdot (n \cdot (n-1)/2)$ façons d'appliquer la règle de résolution.*

Exercice 3.28 *Soit A une formule en CNF et C une clause. Expliquez comment utiliser la méthode de résolution pour établir si l'implication $A \rightarrow C$ est valide.*

Exercice 3.29 *Un exercice de révision. On considère les formules en CNF suivantes :*

1. $\neg x \vee (\neg y \vee x)$
2. $(x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (x \vee \neg y) \wedge \neg x$.
3. $(x \vee y) \wedge (z \vee w) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg w)$.

Pour chaque formule :

1. *Si la formule est valide calculez une preuve de la formule dans le système de Gentzen.*
2. *Si la formule est satisfiable mais pas valide calculez une affectation qui satisfait la formule en utilisant la méthode de Davis-Putnam.*
3. *Si la formule n'est pas satisfiable dérivez la clause vide en utilisant la méthode par résolution.*

4 Diagrammes de décision binaire et applications

Les diagrammes de décision binaire (BDD pour *Binary Decision Diagrams*) sont une représentation des fonctions booléennes. Cette représentation avait déjà été remarquée par Lee en 1959 mais son intérêt algorithmique a été réalisé plus récemment par Bryant en 1986.

Les BDD représentent une fonction booléenne comme un circuit composé de multiplexeurs (if-then-else) et de constantes 0 et 1. Une propriété importante de cette représentation est qu'étant donné un ordre sur les variables, le BDD peut être réduit efficacement à une *forme canonique*. On parle alors de diagramme de décision binaire ordonné et réduit (ROBDD pour *reduced ordered binary decision diagram*). En pratique, la représentation canonique est considérablement plus compacte que la représentation explicite dont la taille est exponentielle dans le nombre de variables de la fonction.

Une deuxième propriété importante est qu'il est possible de manipuler directement les représentations canoniques pour calculer la conjonction, la disjonction, le complémentaire,... La situation est similaire à celle des langages réguliers où un langage peut être représenté par un automate et les opérations d'union, intersection, complémentaire sur les langages peuvent être calculées directement sur les automates. Aujourd'hui, les BDD sont courrament utilisés dans la synthèse et analyse de circuits.

4.1 OBDD

Soit $f : \mathbf{2}^n \to \mathbf{2}$ une fonction booléenne à n variables $x_1, \ldots, x_n$. Si $b \in \{0, 1\}$ est une valeur booléenne, on dénote par $[b/x_i]f : \mathbf{2}^{(n-1)} \to \mathbf{2}$ la fonction booléenne à $n-1$ variables où la variable x_i est remplacée par b. On appelle *restriction* cette opération sur les fonctions.

En utilisant la restriction, on peut exprimer une fonction à n variables comme une combinaison booléenne de fonctions à $n-1$ variables.

$$f = x_i[1/x_i]f + \overline{x_i}[0/x_i]f$$

On nomme cette transformation *expansion de Shannon*. On remarquera que la quantification universelle et existentielle sur une variable propositionnelle s'exprime aussi par le biais de la restriction :

$$\forall x_i \ f = ([1/x_i]f)([0/x_i]f) \quad \exists x_i \ f = ([1/x_i]f) + ([0/x_i]f)$$

Cette transformation entraîne un doublement de la taille de la formule pour chaque quantification.

On abrège l'opérateur ternaire *if_then_else* par $\_ \to \_, \_$. Ainsi :

$$x \to f, f' = (xf) + \overline{x}f'$$

où x est une variable booléenne. On utilise cette notation, pour reformuler l'expansion de Shannon :

$$f = x_i \to [1/x_i]f, [0, x_i]f$$

On fixe un ordre sur les variables, par exemple $x_1 < \cdots < x_n$. On définit par récurrence l'ensemble des expressions qui dépendent d'un sous-ensemble de variables :

- Les expressions 0 et 1 dépendent de l'ensemble vide.
- Si les expressions e_1 et e_2 dépendent de $\{x_{i+1}, \ldots, x_n\}$, alors l'expression $x_i \to e_1, e_2$ dépend de $\{x_i, x_{i+1}, \ldots, x_n\}$.

– Si l'expression e dépend de X et $X \subseteq X'$ alors e dépend de X'.

A partir de la fonction f on peut itérer l'expansion de Shannon en commençant par la variable x_1 et en terminant avec les fonctions constantes 0 et 1. Ainsi on construit :

$$\begin{aligned} f &= x_1 \to [1/x_1]f, [0/x_1]f \\ &= x_1 \to (x_2 \to [1/x_2, 1/x_1]f, [0/x_2, 1/x_1]f), (x_2 \to [1/x_2, 0/x_1]f, [0/x_2, 0/x_1]f) \\ &\cdots \\ &= \cdots \end{aligned}$$

On peut représenter l'expression comme un arbre binaire complet de profondeur $n-1$ où les noeuds internes sont étiquetés par les variables et les noeuds terminaux par 0 ou 1, et les deux arêtes sortantes d'un noeud interne sont étiquetées par 0 et 1. Cette représentation dépend de l'ordre des variables et pour cette raison on parle de BDD *ordonnés* (OBDD).

On remarquera que cette représentation a aussi une taille $O(2^n)$. Cependant, la représentation d'un OBDD comme un arbre binaire est souvent redondante et une représentation plus compacte est possible par partage de sous-arbres communs. Dans ce cas, le BDD est représenté par un *graphe dirigé acyclique* (DAG) connexe et avec une racine.

4.2 Simplification

Soit N un ensemble fini de noeuds et $V = \{x_1, \ldots, x_n\}$ un ensemble de variables ordonné par $x_1 < \cdots < x_n$. On peut représenter un OBDD comme suit :

$$\begin{array}{ll} v : N \to \{0,1\} \cup V & \text{(étiquette des noeuds)} \\ l : N \to (N \cup \{\uparrow\}) & \text{(arête sortant étiqueté par 0)} \\ h : N \to (N \cup \{\uparrow\}) & \text{(arête sortant étiqueté par 1)} \end{array}$$

tel que :

– Le graphe résultat est acyclique et tous les noeuds sont accessibles à partir d'un noeud identifié comme étant la racine.

– Les noeuds non-terminaux sont étiquetés par des variables et les noeuds terminaux sont étiquetés par 0 ou 1. En d'autres termes :

$$v(n) \in V \;\Rightarrow\; l(n), h(n) \in N \qquad v(n) \in \{0,1\} \;\Rightarrow\; l(n) = h(n) = \uparrow$$

– L'ordre des variables est respecté :

$$\begin{array}{ll} v(n) \in V \text{ and } v(l(n)) \in V & \Rightarrow \quad v(n) < v(l(n)) \\ v(n) \in V \text{ and } v(h(n)) \in V & \Rightarrow \quad v(n) < v(h(n)) \end{array}$$

A partir d'un OBDD on applique trois règles de simplification :

– Soient n et n' deux noeuds terminaux distincts avec la même étiquette. Alors tous les pointeurs à n' peuvent être redirigés sur n et n' peut être éliminé.

– Soit n un noeud non-terminal et $l(n) = h(n) = n'$. Alors tous les pointeurs à n peuvent être redirigés sur n', et n peut être éliminé.

– Soient n et n' deux noeuds non-terminaux distincts tels que $v(n) = v(n')$, $l(n) = l(n')$ et $h(n) = h(n')$. Alors tous les pointeurs à n' peuvent être redirigés sur n et n' peut être éliminé.

On écrit $B \mapsto B'$ si un OBDD B est transformé en B' par une des règles de simplification. On dit que B est un forme normale s'il ne peut pas être simplifié.

Théorème 4.1 (1) *Si B est un OBDD bien formé par rapport à un ordre donné et $B \mapsto B'$ alors B' est un OBDD bien formé par rapport au même ordre.*

(2) *Toute séquence de simplification termine.*

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

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

Exercice 4.2 *Calculez le ROBDD pour 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 4.3 (1) *Calculez le ROBDD pour la fonction $(a \wedge b) \vee (c \wedge d)$ avec ordre $a < b < c$.*

(2) *Calculez le ROBDD pour un comparateur de 2-bits $\wedge_{i=1,2}(a_i = b_i)$ en utilisant les ordres $a_1 < b_1 < a_2 < b_2$ et $a_1 < a_2 < b_1 < b_2$.*

(3) *Généraliser à un comparateur de n-bits et déterminez le nombre de noeuds dans le ROBDD pour les ordres $a_1 < b_1 < \cdots < a_n < b_n$ et $a_1 < \cdots a_n < b_1 < \cdots < b_n$.*

Exercice 4.4 *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 ROBDD (BDD ordonné et réduit) qui représente la fonction p par rapport à l'ordre $x_1 < \cdots < x_n$.

Exercice 4.5 *Montrez que la satisfaction et la validité d'une fonction booléenne représentée par un ROBDD peut être décidée en $O(1)$.*

Exercice 4.6 *On sait qu'un langage régulier (ou rationnel) éventuellement infini peut être représenté par un graphe étiqueté fini. On pourrait représenter une fonction $f : \mathbf{2}^n \to \mathbf{2}$ par le langage :*

$$L_f = \{x_1 \ldots x_n \mid f(x_1, \ldots, x_n) = 1\} \subset \{0,1\}^*$$

Comparez l'automate M_f qui reconnaît le langage L_f avec le ROBDD associé à la fonction f. Est-ce-que les deux représentations ont la même taille ?

4.3 Ordre des variables

L'ordre des variables a un effet important sur la taille d'un ROBDD. Par exemple, considérons la fonction $\Sigma_{i=1,\ldots,n} a_i b_i$. Avec l'ordre $a_1 < b_1 < \cdots < a_n < b_n$ la taille du ROBDD est $O(n)$ alors qu'avec l'ordre $a_1 < \cdots a_n < b_1 < \cdots < b_n$ la taille du ROBDD est $O(2^n)$.

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 ROBBD varie entre $O(n)$ et $O(n^2)$.
- Pour le bit central de la fonction d'addition sur n bits la taille varie entre $O(n)$ et $O(2^n)$.

– Pour le bit central de la fonction de multiplication sur n bits la taille est toujours $O(2^n)$.

Exercice 4.7 (1) *Montrez que* $f : \mathbf{2}^n \to \mathbf{2}$ *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 a une représentation comme ROBDD dont la taille est* $O(n^2)$.

4.4 Restriction

Étant donné un OBDD pour la fonction f, le calcul de la restriction, par exemple $[0/x]f$, consiste à rediriger toute arête qui pointe au noeud n tel que $v(n) = x$ vers $l(n)$. Le calcul de $[1/x]f$ est similaire.

Exercice 4.8 *Montrez que l'application de l'algorithme de restriction sur un ROBDD peut ne pas produire un ROBDD.*

4.5 Application

On définit un algorithme A pour l'application qui prend l'OBDD de deux fonctions booléennes $f, g : \mathbf{2}^n \to \mathbf{2}$ et une opération binaire $op : \mathbf{2}^2 \to \mathbf{2}$, et retourne un OBDD pour la fonction $(f\ op\ g) : \mathbf{2}^n \to \mathbf{2}$ (par rapport au même ordre).

La remarque fondamentale est que l'opération op commute avec l'expansion de Shannon :

$$f\ op\ g = x \to ([1/x]f\ op\ [1/x]g), ([0/x]f\ op\ [0/x]g)$$

L'algorithme visite les deux OBDD en profondeur d'abord. En supposant que n_f et n_g soient les racines des deux OBDD, l'appel $A(n_f, n'_g, op)$ retournera la racine de l'OBDD pour $f\ op\ g$. L'algorithme récursif est décrit dans la table 1, où *new* est une fonction qui retourne un nouveau noeud.

Cet algorithme peut être amené à évaluer plusieurs fois le même couple de sous-arbres. Pour éviter cela, on considère une optimisation qui consiste à garder dans un tableau de hachage les couples de sous-arbres déjà visités. Une deuxième optimisation est d'arrêter les appels récursifs chaque fois qu'on arrive à une feuille d'un des sous-arbres 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 3 simplifications. De cette façon, on peut générer directement un ROBDD à partir de ROBDD.

Quand toutes ces optimisations sont mises en oeuvre et étant donné un tableau 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 OBDD qui représentent f et g. En gros, une opération logique peut au plus élever au carré la taille de la représentation. Bryant appelle cela une propriété de *dégradation gracieuse* (bien sûr l'itération d'un carré donne un exponentiel !)

Exercice 4.9 *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 :*

$$x_{n-1} < y_{n-1} < \cdots < x_0 < y_0$$

$A(n, n', op)$	$=$ case
$v(n) = v(n') \in V$	$: n'' := new; v(n'') := v(n);$
	$l(n'') := A(l(n), l(n'), op); h(n'') := A(h(n), h(n'), op); n''$
$v(n) <_V v(n')$	$: n'' := new; v(n'') := v(n);$
	$l(n'') := A(l(n), n', op); h(n'') := A(h(n), n', op); n''$
$v(n') <_V v(n)$	$: n'' := new; v(n'') := v(n');$
	$l(n'') := A(n, l(n'), op); h(n'') := A(n, h(n'), op); n''$
$\{v(n), v(n')\} \subseteq \{0, 1\}$	$: n'' := new; v(n'') := v(n)\ op\ v(n');$
	$l(n'') :=\uparrow; h(n'') :=\uparrow; n''$
$v(n) \in \{0, 1\}$	$: n'' := new; v(n'') := v(n');$
	$l(n'') := A(n, l(n'), op); h(n'') := A(n, h(n'), op); n''$
$v(n') \in \{0, 1\}$	$: n'' := new; v(n'') := v(n);$
	$l(n'') := A(l(n), n', op); h(n'') := A(h(n), n', op); n''$

TAB. 1 – Algorithme pour l'application

1. *Calculez le ROBDD (BDD ordonné et réduit) qui représente la fonction f pour $n = 3$.*
2. *Calculez en fonction de n le nombre de noeuds du ROBDD qui représente la fonction f.*

Exercice 4.10 *Représentez avec une expression booléenne la propriété que deux fonctions $f, g : \mathbf{2}^n \to \mathbf{2}$ ont la même valeur sur toutes les entrées $\vec{x}$ telles que $d(\vec{x}) = 0$.*

Exercice 4.11 *Donnez une méthode pour synthétiser un OBDD directement à partir d'un circuit booléen (sans passer par la table de vérité). Appliquez la méthode à la fonction*

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

4.6 Une application au calcul de l'accessibilité

L'applicabilité des OBDD dépend du fait que les fonctions booléennes peuvent représenter toute construction sur des domaines finis.

- Soit $f : D \to D'$ une fonction, où D et D' sont des domaines finis. Soit $n = \lceil lg_2(\sharp D) \rceil$ et soit $m = \lceil lg_2(\sharp D') \rceil$. Étant donné un codage des éléments de D et D' comme vecteurs de booléens, on peut représenter f comme une fonction booléenne $f' : \mathbf{2}^n \to \mathbf{2}^m$.
- Les sous-ensembles d'un domaine D peuvent être représentés par leur fonction caractéristique qui modulo le codage des éléments en D sont simplement des fonctions booléennes. Les opérations ensemblistes peuvent être calculées directement sur les fonctions caractéristiques.
- Une relation $R \subseteq D \times D'$ se représente comme une fonction caractéristique $\chi_R : D \times D' \to \{0, 1\}$, et donc comme une fonction booléenne.
- Les quantifications existentielles et universelles sur des domaines finis peuvent être calculées sur la représentation OBDD.

– Enfin l'égalité de fonctions, ensembles, relations,... représentés comme ROBDD peut être facilement vérifiée.

Donc en utilisant les fonctions booléennes/les BDD on peut représenter les fonctions finies, les relations finies et les opérations logiques telles que la conjonction, la disjonction, la négation et les quantifications sur des domaines finis.

Nous considérons dans la suite un exemple concret. Soit $M = (\Sigma, Q, q_0, \delta)$ un automate fini (voir section 5). On définit une relation R pour l'accessibilité comme suit :

$$(q, q') \in R \text{ si } \exists\, a \in \Sigma \;\; \delta(a, q) = q'$$

Pour déterminer s'il y a un chemin de q_0 à q, on calcule la clôture réflexive et transitive R^* de R et on vérifie si $(q^o, q) \in R^*$. Le calcul de la clôture réflexive et transitive peut être effectué par une méthode itérée :

$$\begin{array}{ll} R_0 & = R \cup Id \\ R_{n+1} & = R_n \circ R_n \end{array}$$

Exercice 4.12 *Vérifiez que* $\exists\, m \;\; R_m = R_{m+1}$ *et que* $R_m = R_{m+1}$ *implique* $R_m = R^*$.

On présente une méthode pour effectuer ce calcul sur les OBDD.

1. On code les états comme vecteurs de variables booléennes, par exemple $\vec{q} \in \mathbf{2}^n$ et on fixe un ordre sur les variables. On aura besoin de $3n$ variables ordonnées comme suit :

$$q_1 < \ldots < q_n < q'_1 < \ldots < q'_n < q''_1 < \ldots < q''_n$$

2. On construit un OBDD pour la fonction $R : \mathbf{2}^{n+n} \to \mathbf{2}$ telle que $R(\vec{q}, \vec{q''}) = 1$ si et seulement si il y a une arête de l'état $\vec{q}$ à l'état représenté par $\vec{q''}$.
3. On construit un OBDD pour $Id : \mathbf{2}^{n+n} \to \mathbf{2}$ tel que $Id(\vec{q}, \vec{q''}) = 1$ si et seulement si $\vec{q} = \vec{q''}$.
4. On construit un OBDD pour $R_0 : \mathbf{2}^{n+n} \to \mathbf{2}$ par $APPLY(R, Id, or)$.
5. Étant donné $R_n(\vec{q}, \vec{q''})$ on construit $R_n(\vec{q}, \vec{q'})$ et $R_n(\vec{q'}, \vec{q''})$ par renommage des variables. On exprime la composition de relations par :

$$(R_n \circ R_n)(\vec{q}, \vec{q''}) \text{ ssi } \exists\, \vec{q'} \;\; R_n(\vec{q}, \vec{q'}) R_n(\vec{q'}, \vec{q''})$$

   Ainsi R_{n+1} est calculée de R_n en utilisant $APPLY$.
6. Enfin le test de convergence de l'itération est effectué en comparant les ROBDD pour R_n et R_{n+1}.

5 Langages formels et automates finis

Un circuit booléen est un premier exemple de *modèle de calcul*. Nous avons montré que toute fonction $f : \mathbf{2}^n \to \mathbf{2}$ peut être calculée par un circuit booléen. Cependant, on peut remarquer une limitation de tout circuit booléen : la taille de son entrée est figée (un vecteur de n bits). Il est évidemment intéressant de disposer d'un dispositif de calcul plus général capable de traiter des entrées de taille arbitraire. Par exemple, un circuit peut seulement comparer deux entiers d'une certaine taille. Si on veut doubler la taille des entiers en entrée, on sera obligé de construire un nouveau circuit. Or on souhaiterait disposer d'un dispositif capable de comparer deux entiers de taille arbitraire. Ce type de considérations nous mènent vers des modèles de calcul plus généraux. En particulier, dans cette section nous considérons le modèle des *automates finis*. On dénote par $\mathbf{2}^*$ l'ensemble des suites finies de bits. Dans le modèle des automates finis l'entrée peut être vue comme un élément de $\mathbf{2}^*$. Ainsi un automate fini définit (ou calcule) une fonction $f : \mathbf{2}^* \to \mathbf{2}$.[3]

5.1 Notation et problèmes

Un *alphabet* Σ est un ensemble fini et non-vide. On appelle *caractère* un élément de Σ. Si X est un ensemble alors X^* est l'ensemble des séquences finies d'éléments de X :

$$X^* = \{x_1 \dots x_n \mid n \geq 0, x_i \in X\} \tag{4}$$

En particulier, si Σ est un alphabet alors on appelle *mots* les éléments de Σ^* et on dénote avec ϵ la séquence vide.

On associe avec chaque mot $w \in \Sigma^*$ sa *longueur* $|w| \in \mathbf{N}$ en définissant $|\epsilon| = 0$ et $|aw| = 1 + |w|$ si $a \in \Sigma$ et $w \in \Sigma^*$.

Si $w_1, w_2 \in \Sigma^*$ alors on dénote par $w_1 w_2 \in \Sigma^*$ la *concaténation* de mots. On remarque que l'opération de concaténation est associative et que le mot vide ϵ est une identité gauche et droite.

Un *langage formel* L sur un alphabet Σ est simplement un sous-ensemble de Σ^*. L'opération de concaténation est étendue aux langages de la façon suivante : si $L_1, L_2 \subseteq \Sigma^*$ alors

$$L_1 L_2 = \{w_1 w_2 \mid w_i \in L_i, i = 1, 2\} \ .$$

La concaténation de langages est aussi une opération associative ayant le langage $\{\epsilon\}$ comme identité gauche et droite.

L'*itération* L^* d'un langage L est définie par :

$$L^0 = \{\epsilon\} \quad L^{n+1} = LL^n \quad L^* = \textstyle\bigcup_{n \in \mathbf{N}} L^n \ .$$

En particulier, on remarque que $\emptyset^* = \{\epsilon\} \neq \emptyset$. On définit aussi L^+ comme $L^+ = LL^*$.

Dans la théorie des langages formels, on s'intéresse aux problèmes suivants :

- Définir des outils formels (automates, grammaires,...) qui *décrivent* de façon synthétique mais précise un langage formel.
- Étant donné un langage L sur un alphabet Σ, construire un programme (un automate) qui *décide* si un mot w appartient à L.

[3] Dans la suite du cours, on verra qu'une grande partie des fonctions $f : \mathbf{2}^* \to \mathbf{2}$ ne sont pas calculables par un automate fini ou, plus en général, par un 'programme'.

– Classifier les langages selon la *complexité* du langage de spécification et/ou de l'automate qui les reconnaît.
– Développer des méthodes pour montrer qu'un certain langage n'est pas dans une certaine classe.

Ici on se limite à définir une famille de programmes (les automates finis) qui décident la classe des langages réguliers (ou rationnels).

5.2 Automates finis déterministes

Définition 5.1 (AFD) *Un automate fini déterministe (AFD) M est un vecteur $(\Sigma, Q, q_o, F, \delta)$ où Σ est un alphabet, Q est un ensemble fini qui représente l'ensemble des états de l'automate, $q_o \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux (ou accepteurs), et $\delta : \Sigma \times Q \to Q$ est la fonction de transition.*

Un AFD se représente graphiquement comme un graphe dirigé tel que : (i) les noeuds correspondent aux états, (ii) il y a une arête étiquetée par a de q à q' si et seulement si $\delta(a, q) = q'$, (iii) le noeud qui correspond à l'état initial est marqué par $>$ et (iv) les noeuds qui correspondent aux états finaux ont un double contour.

Dans la suite, on procède en trois étapes :

1. On définit la notion de *configuration* d'un automate.
2. On décrit comment un automate peut se déplacer d'une configuration à une autre.
3. On spécifie quels mots sont acceptés par l'automate.

Une méthodologie similaire est utilisée dans la suite pour un type d'automate plus général qu'on appelle *Machine de Turing*.

Définition 5.2 *Soit $M = (\Sigma, Q, q_o, F, \delta)$ un AFD. Une configuration est un couple $(w, q) \in \Sigma^* \times Q$. On définit une relation de réduction $\vdash_M$ par $(aw, q) \vdash_M (w, \delta(a, q))$ et on suppose que $\vdash_M^*$ est la clôture réflexive et transitive de $\vdash_M$. Le langage $\mathcal{L}(M)$ reconnu (ou accepté) par M est défini par :*

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid (w, q_o) \vdash_M^* (\epsilon, q) \text{ and } q \in F\} \ .$$

Exemple 5.3 *Soit $M = (\{a, b\}, \{1, 2\}, 1, \{2\}, \delta)$ avec fonction de transition δ spécifiée comme suit :*

État	*Entrée*	
	a	b
1	1	2
2	1	2

Il n'est pas difficile de montrer que $\mathcal{L}(M)$ est l'ensemble des mots qui terminent par b.

Remarque 5.4 *Dans la définition de AFD on insiste pour que pour chaque état q et pour chaque caractère a de l'alphabet il y ait exactement* une arête *sortante de q avec étiquette a. En pratique, on peut relâcher cette condition et demander juste qu'il y ait* au plus une arête *sortante de q avec étiquette a. Un tel automate peut être transformé facilement en un AFD en introduisant un état 'puits' q_s et en étendant la fonction de transition δ de façon telle que $\delta(a, q_s) = q_s$ pour tout $a \in \Sigma$ et $\delta(a, q) = q_s$ chaque fois que $\delta(a, q)$ n'est pas défini.*

Remarque 5.5 (minimisation) *Il est facile de construire différents AFD qui acceptent le même langage. Cependant on peut montrer que parmi ces automates il y en a un qui a un nombre minimum d'états. De plus cet automate est unique à renommage des états près.*

5.3 Automates non-déterministes

Nous considérons trois extensions de la notion d'AFD qui nous mènent à la notion d'automate fini non-déterministe (AFN).

1. On permet de lire plus qu'un caractère dans un pas de calcul.
2. On permet de ne pas lire un caractère (ϵ-transition).
3. Pour un noeud donné, on autorise deux ou plus arêtes sortantes étiquetées avec le même mot.

Définition 5.6 (AFN) *Un automate fini non-déterministe (AFN) N est un vecteur $(\Sigma, Q, q_o, F, \delta)$ où Σ est un alphabet, Q est un ensemble fini d'états, q_o est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux et $\delta : Q \times \Sigma^* \to 2^Q$ est une fonction de transition qui s'évalue dans l'ensemble vide presque partout.*

Une configuration *pour un AFN est un couple $(w, q) \in \Sigma^* \times Q$. La* relation de réduction $\vdash_N$ *est définie par :*

$$(w, q) \vdash_N (w', q') \text{ si } w = w''w' \text{ et } q' \in \delta(w'', q)$$

et le langage reconnu $\mathcal{L}(N)$ *est défini par*

$$\mathcal{L}(N) = \{w \in \Sigma^* \mid (w, q_o) \vdash_N^* (\epsilon, q) \text{ et } q \in F\} \ .$$

Dans un AFD, étant donné un mot w on trouve un chemin de calcul unique qui va de (w, q_o) à (ϵ, q), pour un certain q. Par opposition, dans un AFN on peut avoir plusieurs chemins, et le w est accepté si *au moins un chemin* mène à un état final. Un problème fondamental est de comprendre si et dans quel mesure le calcul non-déterministe est plus puissant que le calcul déterministe.

Théorème 5.7 (déterminisation) *Pour tout AFN on peut construire un AFD qui accepte le même langage.*

Proof hint. (1) Si un automate peux exécuter le pas de calcul $(a_1 \cdots a_n, q) \vdash (\epsilon, q')$ avec $n \geq 2$ alors on introduit $n-1$ nouveaux états non-finaux $q_1, \ldots, q_{n-1}$ et on redéfinit la fonction de transition pour que :

$$(a_1 \cdots a_n, q) \vdash (a_2 \cdots a_n, q_1) \vdash \cdots \vdash (a_n, q_{n-1}) \vdash (\epsilon, q') \ .$$

(2) On peut donc supposer que dans une transition un automate $N = (\Sigma, Q, q_o, F, \delta)$ lit au plus un caractère et que la fonction de transition a le type $\delta : (\Sigma \cup \{\epsilon\}) \times Q \to 2^Q$. Maintenant, l'idée est d'éliminer les ϵ-transitions, en ajoutant une transition étiquetée par a de q à q_1, chaque fois qu'il y a un chemin de q à q_1 dont toutes les arêtes sont étiquetées par ϵ sauf une qui est étiquetée par a.

Formellement, on introduit une notion de ϵ-clôture d'un état q comme suit :

$$E(q) = \{q' \mid (\epsilon, q) \vdash^* (\epsilon, q')\} \ .$$

Ensuite on construit un nouveau automate $N' = (\Sigma, Q, q_o, F', \delta')$ où $F' = \{q \in Q \mid E(q) \cap F \neq \emptyset\}$ et $\delta' : \Sigma \times Q \to 2^Q$ est définie par

$$\delta'(a, q) = \bigcup_{q' \in E(q)} \{E(q'') \mid q'' \in \delta(a, q')\} \ .$$

Dans d'autres termes, $(a, q) \vdash_{N'} (\epsilon, q_1)$ ssi

$$(a, q) \vdash_N^* (a, q') \vdash_N (\epsilon, q'') \vdash_N^* (\epsilon, q_1) \ .$$

(3) On peut supposer que l'automate N a une fonction de transition δ avec le type suivant $\delta : \Sigma \times Q \to 2^Q$. Supposons que de l'état q, en lisant a, l'automate peut aller ou bien dans q_1 ou bien dans q_2, *c.-a.-d.*, $\delta(a, q) = \{q_1, q_2\}$. Pour simuler ce comportement non-déterministe avec un AFD M on dit que M placé dans l'état q, en lisant a, peut aller dans un 'nouveau état' $\{q_1, q_2\}$ qui est capable de 'simuler' le comportement à la fois de q_1 et q_2.

Formellement, on construit un AFD $M = (\Sigma, 2^Q, \{q_o\}, F_M, \delta_M)$ dont les états sont des sous-ensembles de l'ensemble des états de N et tel que :

$$\begin{array}{ll} F_M & = \{X \subseteq Q \mid X \cap F \neq \emptyset\} \\ \delta_M(a, X) & = \bigcup_{q \in X} \delta(a, q) \ . \end{array}$$

•

Exemple 5.8 *Considérons l'AFN* $N = (\{a, b\}, \{1, 2, 3\}, 1, \{2\}, \delta)$ *avec*

$$\delta(\epsilon, 1) = \{2\} \quad \delta(bb, 1) = \{3\} \quad \delta(a, 2) = \{2\} \quad \delta(\epsilon, 3) = \{1\} \quad \delta(a, 3) = \{3\} \ .$$

On élimine la transition étiquetée par bb *en introduisant un état auxiliaire, ensuite on élimine les* ϵ*-transitions, et enfin on déterminise l'automate.*

Remarque 5.9 (coût) *Il y a des AFN tels que chaque AFD équivalent a un nombre d'états qui est exponentiel dans le nombre d'états de l'AFN.*

Remarque 5.10 (langages réguliers) *On dit qu'un langage accepté par un automate fini est régulier (ou rationnel). La classe des langages réguliers a une théorie très riche qui sera l'objet d'un cours au deuxième semestre.*

Exercice 5.11 *Montrez que pour tout langage* L, $L^* = (L^*)^*$.

Exercice 5.12 *Montrez qu'il existe des langages* L_1 *et* L_2 *tels que* $(L_1 \cup L_2)^* \neq L_1^* \cup L_2^*$.

Exercice 5.13 *Montrez qu'il existe des langages* L_1 *et* L_2 *tels que* $(L_1 \cdot L_2)^* \neq L_1^* \cdot L_2^*$.

Exercice 5.14 *Considérons l'automate fini* $M = (Q, \Sigma, \delta, q_0, F)$, *où* $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$ *et la fonction* δ *est définie par le tableau suivant :*

État	*Entrée*	
	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Vérifiez si les chaînes 1011010 *et* 101011 *sont acceptées par* M. *Prouvez que* $\mathcal{L}(M)$ *est l'ensemble des mots composés d'un nombre pair de* 0 *et d'un nombre pair de* 1.

Exercice 5.15 *Pour chacun des langages suivants, construire un automate fini non déterministe qui l'accepte :*

1. *Les représentations binaires des nombres pairs.*
2. *Les représentations décimales des multiples de* 3.
3. *Le langage des mots sur l'alphabet* $\{a, b\}$ *contenant ou bien la chaîne aab ou bien la chaîne aaab.*
4. *Le langage des mots sur l'alphabet* $\{0, 1\}$ *dont le troisième caractère de droite existe et est égale à* 1.

Construire des automates déterministes pour les langages décrits ci-dessus.

Exercice 5.16 *Soient M un AFD qui accepte un langage L et N_1, N_2 deux AFN qui acceptent les langages L_1, L_2, respectivement (sur un alphabet Σ fixé).*

1. *Montrez qu'on peut construire un AFD qui accepte le langage complémentaire $\Sigma^* \backslash L$.*
2. *Montrez qu'on peut construire un AFN qui accepte le langage $L_1 \cup L_2$ et le langage itéré $(L_1)^*$.*
3. *Conclure que la classe des langages acceptés par un AFD est stable par union, intersection, complémentaire et itération.*

Exercice 5.17 *Soit l'automate fini non-déterministe $M = (Q, \Sigma, \delta, q_0, F)$, où $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$, $F = \{q_0, q_2\}$, et la fonction de transition δ est définie par le tableau suivant :*

État	*Entrée*		
	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$

Transformez cet automate en automate fini déterministe.

Exercice 5.18 *Transformez l'automate $M = (Q, \Sigma, \delta, q_0, F)$ suivant en automate fini déterministe. On suppose que $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$, $F = \{q_2\}$, et la fonction de transition δ est définie par le tableau suivant :*

État	*Entrée*			
	0	1	2	ϵ
q_0	$\{q_0\}$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

6 Calculabilité

Certains problèmes calculatoires demandent une mémoire qui est fonction de la taille de l'entrée (par exemple le tri d'une liste d'éléments ou la multiplication de deux matrices). De tels problèmes ne peuvent pas être résolus par des automates finis dont la mémoire est bornée *a priori*. On considère le problème de formaliser un modèle de calcul suffisamment général pour calculer tout ce qu'un 'ordinateur' pourrait calculer en disposant d'une quantité illimitée de temps et de mémoire. Plusieurs modèles *équivalents* ont été proposés à partir des années '30. On base la présentation sur les *machines de Turing* (MdT) qui peuvent être vues comme une simple généralisation des automates finis.

6.1 Machines de Turing

Un automate fini dispose d'un contrôle fini et d'un ruban sur lequel il peut déplacer sa tête de lecture de gauche à droite. Une machine de Turing a en plus la possibilité d'écrire sur le ruban et de déplacer la tête de lecture de droite à gauche.

Définition 6.1 *Une machine de Turing (déterministe) M est un vecteur $M = (Q, \Sigma, \Gamma, \sqcup, q_0, q_a, q_r, \delta)$ où :*

- *Q est un ensemble fini d'états.*
- *Σ est l'alphabet d'entrée.*
- *Γ est l'alphabet du ruban.*
- *$\sqcup \in \Gamma \backslash \Sigma$ est un symbole spécial,*
- *$q_0, q_a, q_r \in Q$ sont des états. En particulier q_0 est l'état initial et q_a, q_r sont deux états finaux distincts qui entraînent l'arrêt du calcul.*
- *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ est la fonction (déterministe) de transition où L pour* left *et R pour* right *sont deux symboles.*

Une configuration *de la machine M est un mot wqw' où $w, w' \in \Gamma^*$ et $q \in Q$. Une* configuration initiale *est un mot $q_0 w$ où $w \in \Sigma^*$ représente l'entrée de la machine.*

Une MdT calcule sur un ruban dont la taille n'est pas bornée à droite. Soit $\sqcup^\omega$ le mot infini $\sqcup \sqcup \sqcup \cdots$ Une configuration wqw' décrit : (i) le contenu du ruban qui est $ww'\sqcup^\omega$, (ii) l'état q de la machine et (iii) la position de la tête de lecture qui lit le premier caractère du mot $w'\sqcup^\omega$.[4]

Un *pas de calcul* est décrit par la fonction δ. En fonction de l'état courant et du symbole en lecture, la machine se déplace dans un nouvel état, écrit un symbole à la place du symbole lu et déplace la tête de lecture à gauche ou à droite. Le déplacement de la tête de lecture à gauche est impossible si le mot w de la configuration courante est vide. Dans ce cas la tête de lecture reste sur place.

[4] Remarquez que les configurations $wqw', wqw'\sqcup, wqw' \sqcup \sqcup, \cdots$ sont équivalentes dans le sens qu'elles décrivent la même situation.

Pour formaliser ces idées, on définit une relation binaire $\vdash_M$. En supposant que $q \notin \{q_a, q_r\}$, la relation $\vdash_M$ est la plus petite relation sur les configurations qui satisfait :

$$\begin{array}{ll}
wqaw' \vdash_M wbq'w' & \text{si } \delta(q,a) = (q', b, R) \\
wq \vdash_M wbq' & \text{si } \delta(q, \sqcup) = (q', b, R) \\
wcqaw' \vdash_M wq'cbw' & \text{si } \delta(q,a) = (q', b, L) \\
qaw' \vdash_M q'bw' & \text{si } \delta(q,a) = (q', b, L) \\
wcq \vdash_M wq'cb & \text{si } \delta(q, \sqcup) = (q', b, L) \\
q \vdash_M q'b & \text{si } \delta(q, \sqcup) = (q', b, L)
\end{array}$$

On remarque que, la fonction δ étant totale, le calcul de M s'arrête si et seulement si la machine arrive à un état final.

Exercice 6.2 *Examinez la définition de machine de Turing et répondez aux questions suivantes :*

1. *Une MdT peut-elle écrire le symbole $\sqcup$ sur le ruban ?*
2. *L'alphabet d'entrée et du ruban peuvent-ils être égaux ?*
3. *La tête de lecture peut-elle rester au même endroit pendant deux étapes consécutives ?*
4. *Une MdT peut-elle contenir un seul état ?*

Un automate fini peut accepter ou refuser un mot, une MdT peut aussi boucler. Dans la définition de langage accepté par une MdT il faut prendre en compte cette troisième possibilité.

Définition 6.3 (1) *Un ensemble $L \subseteq \Sigma^*$ est semi-décidable s'il existe une MdT M telle que $L = \{w \mid q_0 w \vdash_M^* w' q_a w''\}$. Dans ce cas on dit que M semi-décide (ou accepte) L.*

(2) *Un ensemble L est décidable s'il existe une MdT M dont le calcul termine toujours et qui semi-décide L. Dans ce cas on dit que M décide L.*

Exemple 6.4 *On construit une MdT qui décide $\{a^n b^m \mid n, m \geq 0\}$. On a $\Sigma = \{a, b\}$, $\Gamma = \Sigma \cup \{\sqcup\}$ et $Q = \{q_0, q_a, q_r, q_1\}$. On remarque qu'il est inutile de spécifier le comportement de la fonction δ sur les états q_a et q_r car par définition la MdT s'arrête quand elle arrive à ces états. Par ailleurs, il est aussi inutile de spécifier le caractère écrit et le déplacement effectué par la tête de lecture pour toute transition qui va dans les états finaux. En effet, pour les problèmes de décision on s'intéresse seulement à l'état final et on ignore le contenu du ruban et la position de la tête de lecture. Enfin, on peut interpréter l'absence de spécification comme une transition dans l'état q_r. Avec ces conventions, on peut décrire le comportement de la fonction δ par le tableau :*

	a	b	$\sqcup$
q_0	q_0, a, R	q_1, b, R	q_a, –, –
q_1	–	q_1, b, R	q_a, –, –

Comme dans les automates finis, on peut introduire une notation graphique. *Par exemple, on écrira :*

$$q \overset{a/b,L}{\rightarrow} q'$$

pour signifier que la MdT dans l'état q et en lisant a, écrit b, se déplace à gauche (L) et va dans l'état q'.

On remarquera que dans ce cas notre MdT se comporte comme un automate fini : elle se déplace seulement à droite et elle ne modifie pas le contenu du ruban.

Exemple 6.5 *On construit une MdT qui décide $\{a^n b^n \mid n \geq 0\}$. On a $\Sigma = \{a, b\}$, $\Gamma = \Sigma \cup \{X, Y, \sqcup\}$ et $Q = \{q_0, q_a, q_r, q_1, q_2, q_3, q_4\}$. La fonction δ est spécifiée comme suit :*

	a	b	X	Y	$\sqcup$
q_0	q_1, X, R	–	–	–	$q_a, –, –$
q_1	q_1, a, R	q_2, Y, L	–	q_1, Y, R	–
q_2	q_2, a, L	–	q_3, X, R	q_2, Y, L	–
q_3	q_1, X, R	–	–	q_4, Y, R	–
q_4	–	–	–	q_4, Y, R	$q_a, –, –$

Exemple 6.6 *Soit $\Sigma = \{0, 1, \sharp\}$ et $L = \{w\sharp w \mid w \in \{0,1\}^*\}$. On peut construire une MdT qui décide L en prenant $\Gamma = \Sigma \cup \{\sqcup, X\}$. La machine lit le premier caractère b de w, le remplace par X, puis déplace sa tête de lecture à droite pour vérifier que le premier symbole à droite de $\sharp$ est b, le remplace par X, puis revient à gauche du $\sharp$ et ainsi de suite. Un observateur qui regarderait le contenu du ruban verrait par exemple :*

$$01\sharp01\sqcup^\omega \quad X1\sharp01\sqcup^\omega \cdots \quad X1\sharp X1\sqcup^\omega \cdots \quad XX\sharp X1\sqcup^\omega \cdots \quad XX\sharp XX\sqcup^\omega$$

Exercice 6.7 *Donnez la description formelle d'une MdT qui décide le langage $\{w\sharp w \mid w \in \{0,1\}^*\}$.*

Exercice 6.8 (programmation MdT) *Présentez le graphe de transition d'une MdT M déterministe avec alphabet d'entrée $\Gamma = \{0, 1, (0,0), (0,1), (1,0), (1,1)\}$ qui a la propriété suivante : à partir de la configuration initiale $q_0(x_{n-1}, y_{n-1}) \cdots (x_0, y_0)$, M va parcourir l'entrée de gauche à droite et la remplacer par $z_{n-1} \cdots z_0$ où $(z_{n-1} \cdots z_0)_2 = max\{(x_{n-1} \cdots x_0)_2, (y_{n-1} \cdots y_0)_2\}$ et s'arreter dans un état accepteur q_a. En d'autres termes, M doit calculer le maximum des entrées.*

Exercice 6.9 *On se propose de programmer une Machine de Turing avec alphabet d'entrée $\Sigma = \{0, 1, \sharp\}$ qui a la propriété suivante : à partir d'une configuration initiale $q_0\sharp w$ où w est un mot fini composé de 0 et 1 la machine s'arrête dans un état accepteur q_a avec un ruban qui contient le mot $\sharp\sharp w$. En d'autres termes, la fonction de la machine est de décaler d'une case vers la droite le mot w en insérant le symbole $\sharp$ dans la case qui est ainsi libérée.*

1. *Donnez la représentation graphique d'une Machine de Turing qui implémente la fonction de décalage décrite ci-dessus.*
   Suggestion *Il est possible de programmer cette tâche avec une MdT dont la tête de lecture se déplace toujours à droite.*

2. *Tracez le calcul de la machine de la configuration initiale $q_0\sharp10$ à la configuration finale.*

Exercice 6.10 *Donnez la description formelle d'une MdT qui décide le langage des mots sur l'alphabet $\{0\}$ dont la longueur est une puissance de 2 : 2^0, 2^1, 2^2, ...*

Exercice 6.11 *Décrivez informellement une MdT qui décide le langage :*

$$\{a^i b^j c^k \mid i \cdot j = k \text{ et } i, j, k \geq 1\} .$$

Si un calcul termine on peut aussi voir le 'contenu du ruban' comme le *résultat du calcul*. Plus précisément on considère comme 'résultat du calcul' la concaténation de tous les symboles dans l'alphabet d'entrée qui sont sur le ruban à la fin du calcul. Par exemple, si le ruban a la forme $\sqcup a \sqcup \sqcup ba \sqcup^\omega$ et a, b sont des symboles de l'alphabet d'entrée, le résultat du calcul est aba. On écrit $M(w) \downarrow$ si la MdT M avec entrée w termine et $M(w) = w'$ si $M(w) \downarrow$ avec résultat w'.

Définition 6.12 (1) *Une fonction partielle $f : \Sigma^* \rightharpoonup \Sigma^*$ est une fonction* partielle récursive *s'il existe une MdT M avec alphabet d'entrée Σ telle que $f(w) = w'$ si et seulement si $M(w) = w'$.*

(2) *Une fonction* récursive *est une fonction partielle récursive totale, c'est-à-dire qui est définie sur chaque entrée.*

Exercice 6.13 *Soit $\Sigma = \{0, 1\}$ et $suc : \Sigma^* \to \Sigma^*$ la fonction 'successeur' en base 2 telle que :*

$$(suc(w))_2 = (w)_2 + 1$$

Montrez que suc est récursive.

6.2 Énumérations

Une variété de structures finies comme arbres, graphes, polynômes, grammaires, MdT,... peuvent être codées comme mots finis d'un alphabet fini.

Exemple 6.14 (problèmes et langages) *Un graphe dirigé fini est un couple (N, A) où N est un ensemble fini de noeuds et $A \subseteq N \times N$ est un ensemble d'arêtes. Deux graphes dirigés (N, A) et (N', A') sont isomorphes s'il existe une bijection $f : N \to N'$ telle que $(n, n') \in A$ ssi $(f(n), f(n')) \in A'$. Notre objectif est de fixer un alphabet fini Σ et de représenter les graphes dirigés comme un langage sur cet alphabet fini. Plus précisément on va représenter les graphes dirigés à 'isomorphisme près'. Ceci est justifié par le fait qu'en général on s'intéresse aux propriétés des graphes qui sont invariantes par isomorphisme (connectivité, diamètre, isomorphisme,...). On suppose que l'ensemble des noeuds N est un segment initial des nombres naturels codés en binaire, par exemple $N = 0, 1, 10, 11$. En conséquence, A est maintenant un ensemble de couples de nombres naturels codés en binaire. On peut ajouter un symbole $\sharp$ qui agit comme un séparateur. Maintenant le graphe $(\{0, 1, 2, 3\}, \{(2, 0), (1, 3), (2, 3)\})$ peut être représenté par le mot fini sur l'alphabet $\Sigma = \{0, 1, \sharp\}$:*

$$\sharp 0 \sharp 1 \sharp 10 \sharp 11 \sharp\sharp 10 \sharp 0 \sharp 1 \sharp 11 \sharp 10 \sharp 11 \sharp$$

Par le biais de ce codage, on peut considérer à isomorphisme près l'ensemble des graphes dirigés comme un certain langage de mots finis sur un alphabet fini. Si G est un graphe dirigé, on dénote par $\langle G \rangle$ son codage. Supposons maintenant qu'on s'intéresse au problème de savoir si deux graphes dirigés sont isomorphes.[5] *On peut reformuler ce problème comme le problème de la reconnaissance du langage :*

$$L = \{\langle G \rangle \sharp\sharp\sharp \langle G' \rangle \mid G \text{ et } G' \text{ sont isomorphes}\}$$

[5]Notez qu'on peut avoir plusieurs codages qui représentent le même graphe à isomorphisme près.

Exemple 6.15 (fixer un alphabet) *On applique maintenant la même méthode aux MdT. Une MdT est un programme. Il est clair que le 'nom' des états n'affecte pas le comportement d'une MdT. Ainsi on peut supposer que les états sont codés, par exemple, en binaire. Considérons maintenant l'ensemble* Γ*. Il est possible de simuler le comportement d'une MdT* M *qui utilise un alphabet* Γ *avec une autre MdT* M' *qui utilise seulement un alphabet* $\{0, 1, \sqcup\}$*. Si* Γ *a* n *éléments on code chaque élément de* Γ *par une suite binaire de longueur* $k = \lceil lg\ n \rceil$*. Pour simuler un pas de calcul de* M*,* M' *doit : (i) lire* k *symboles consécutifs et en fonction de ces* k *symboles et de l'état courant (ii) écrire* k *symboles et (iii) déplacer la tête de lecture de* k *symboles à droite ou à gauche. Donc, à un codage près, le comportement de toute MdT qui opère sur un alphabet arbitraire peut être simulé par une MdT qui opère sur un alphabet fini qui est fixé une fois pour toutes.*

Exemple 6.16 (énumération de MdT) *On s'intéresse maintenant à la représentation comme mots finis des MdT sur un alphabet donné. On peut fixer un codage pour le symbole* $\sqcup$*, pour les états* q_0, q_a, q_r *et pour les symboles* L, R*. Ensuite, la fonction* δ *peut être représentée en listant son graphe (on peut éventuellement ajouter un symbole spécial pour séparer les différents éléments de la liste comme on l'a fait dans le cas des graphes). En procédant de la sorte toute MdT est représentée par un mot fini sur un alphabet fini. Soit* $MdT(\Sigma) \subseteq \Sigma^*$ *l'ensemble des codages de MdT sur l'alphabet* Σ *choisi. Les mots qui composent cet ensemble doivent représenter comme une liste la fonction* δ *d'une MdT. Il est donc décidable de savoir si un mot appartient à* $MdT(\Sigma)$*. Par ailleurs, on peut définir une fonction récursive et surjective* $\varphi : \Sigma^* \to MdT(\Sigma)$*. Soit* w_0 *le codage d'une MdT. La fonction* φ *est définie par :*

$$\varphi(w) = \begin{cases} w & \text{si } w \text{ code une MdT} \\ w_0 & \text{autrement} \end{cases}$$

Mots ou nombres ? On a étudié la calculabilité de langages de *mots finis*. Une autre possibilité aurait été de considérer la calculabilité de sous-ensembles de *nombres naturels*. La théorie n'est pas vraiment affectée par ce choix car les mots finis peuvent être codés par des nombres naturels et le codage est effectivement calculable comme on va le montrer dans les exercices qui suivent.

Exercice 6.17 *On peut énumérer les couples de nombres naturels en procédant 'par diagonales' :*

$$(0,0), \quad (1,0),(0,1), \quad (2,0),(1,1),(0,2), \quad (3,0)\ldots$$

Montrez que la fonction $\langle m, n\rangle = (m+n)(m+n+1)/2 + n$ *est une bijection entre* $\mathbf{N} \times \mathbf{N}$ *et* $\mathbf{N}$*. Décrire un algorithme pour calculer la fonction inverse.*

Exercice 6.18 *On définit les fonctions* $\langle \_ \rangle_k : \mathbf{N}^k \to \mathbf{N}$ *pour* $k \geq 2$ *:*

$$\begin{array}{ll} \langle m, n\rangle_2 & = \langle m, n\rangle \\ \langle n_1, \ldots, n_k\rangle_k & = \langle \langle n_1, \ldots, n_{k-1}\rangle_{k-1}, n_k\rangle \text{ si } k \geq 3 \end{array}$$

Montrez que les fonctions $\langle \_ \rangle_k$ *sont des bijections.*

Exercice 6.19 *On considère l'ensemble* $\mathbf{N}^*$ *des mots finis de nombres naturels. Notez que* $\mathbf{N}^*$ *est en correspondance bijective avec* $\bigcup_{k \geq 0} \mathbf{N}^k$*. Définissez une bijection entre* $\mathbf{N}^*$ *et* $\mathbf{N}$*.*

Exercice 6.20 *Soit $\Sigma = \{a, b, \ldots, z\}$ un alphabet fini. On peut énumérer les éléments de Σ^* comme suit :*

$$\epsilon, \quad a, b, \ldots, z, \quad aa, \ldots, az, ba, \ldots, bz, za, \ldots, zz, \quad aaa, \ldots$$

Si Σ contient k éléments on aura k^0 mots de longueur 0, k mots de longueur 1, k^2 mots de longueur 2, ... Définissez une bijection entre Σ^* et $\mathbf{N}$.

MdT universelle Un corollaire de ces exercices est qu'il y a une bijection $\langle \_, \_ \rangle : \Sigma^* \times \Sigma^* \to \Sigma$. Par le biais de cette bijection, une MdT peut interpréter tout mot w comme un couple de mots $\langle w_1, w_2 \rangle$. Par ailleurs, par le biais de la fonction φ une MdT peut interpréter tout mot comme le codage d'une MdT.

On peut alors construire une MdT U qu'on appelle *MdT universelle* telle que

$$U(\langle w_1, w_2 \rangle) = \varphi(w_1)(w_2)$$

La machine U –dont on omet les détails de construction– reçoit un mot w qui est interprété comme un couple de mots w_1, w_2. Ensuite le mot w_2 est interprété comme l'entrée de la MdT décrite par le premier mot w_1. La MdT U simule la MdT $\varphi(w_1)$ sur l'entrée w_2. Ainsi, la machine U se comporte comme un *interprète* qui reçoit en argument un programme et une entrée et calcule le résultat du programme sur l'entrée.

Exercice 6.21 (1) *Montrez qu'un langage est semi-décidable si et seulement si il est le domaine de définition d'une fonction partielle récursive.*

(2) *On dit qu'un langage $L \subseteq \Sigma^*$ est* récursivement énumérable *s'il est l'image d'une fonction partielle récursive. Montrez qu'un langage L est récursivement énumérable si et seulement si il est semi-décidable.*

Suggestion : Soit M une MdT et $w_0, w_1, w_2, \ldots$ une suite d'entrées. On peut simuler M sur w_0 pour 0 pas, sur w_0 pour 1 pas, sur w_1 pour 0 pas, sur w_0 pour 2 pas, sur w_1 pour 1 pas, sur w_2 pour 0 pas,...

Exercice 6.22 (1) *Rappel : tout nombre naturel $n \geq 2$ admet une décomposition unique comme produit $p_1^{n_1} \cdots p_k^{n_k}$ où $k \geq 1$, $p_1 < \cdots < p_k$ sont des nombres premiers et $n_1, \ldots, n_k \geq 1$. En utilisant ce fait, définissez une fonction surjective de $\mathbf{N}$ dans les parties finies de $\mathbf{N}$.*

(2) *On ne peut pas généraliser aux parties de $\mathbf{N}$! Supposez une énumération $e : \mathbf{N} \to 2^{\mathbf{N}}$. Considérez $X = \{n \mid n \notin e(n)\}$. Comme e est surjective, il existe n_X tel que $e(n_X) = X$ et soit $n_X \in X$ soit $n_X \notin X$. Montrez que dans les deux cas on arrive à une contradiction.*

(3) *On dit qu'un ensemble X est dénombrable s'il y a une fonction bijective entre X et les nombres naturels $\mathbf{N}$.*

(3.1) *Montrez que l'ensemble des langages sur un alphabet Σ n'est pas dénombrable.*

(3.2) *Conclure qu'il y a des langages qui ne sont pas semi-décidables.*

On résume ces considérations comme suit :

- Un problème algorithmique peut être (souvent) reformulé comme un problème de reconnaissance d'un langage.
- Sans perte de généralité, nous pouvons limiter notre attention aux MdT qui opèrent sur un alphabet Γ fixé une fois pour toutes.

– On peut coder une MdT comme un mot fini et on peut énumérer tous les codages de MdT sur un alphabet donné.
– A un codage près, il y a autant de MdT que de nombres naturels alors que l'ensemble des langages a la cardinalité des parties de nombres naturels. Il doit donc y avoir des langages qui ne sont pas décidables.
– On peut s'intéresser de façon équivalente à la calculabilité de langages de mots finis, d'ensembles de couples de mots finis, d'ensembles de nombres naturels,...
– On peut construire une MdT *universelle* qui reçoit en entrée le codage d'une MdT M et une entrée w et simule le calcul de M sur w.

6.3 Temps de calcul

Un pas de calcul d'une MdT est une opération élémentaire qui demande un effort de calcul borné : il s'agit de consulter un tableau fini, d'écrire un symbole et de déplacer d'une position la tête de lecture. Il semble donc raisonnable de mesurer le *temps de calcul* d'une MdT simplement comme le nombre de pas de calcul nécessaires pour arriver à un état final.

Définition 6.23 *Soit M une MdT qui termine sur toute entrée. La complexité en temps de M est une fonction $t : \mathbf{N} \to \mathbf{N}$ où $t(n)$ est le nombre maximal de pas de calcul nécessaires à la machine pour terminer sur une entrée de taille n (la taille d'un mot est sa longueur).*

Souvent on s'intéresse seulement à l'ordre de grandeur de la complexité.

Définition 6.24 *Soient $f, g : \mathbf{N} \to \mathbf{N}$ deux fonctions sur les nombres naturels. On dit que f est $O(g)$ s'ils existent $n_0, c \in \mathbf{N}$ tels que pour tout $n \geq n_0$, $f(n) \leq cg(n)$.*

En d'autres termes, f est $O(g)$ si presque partout f est dominée par g à une constante multiplicative près.

Exercice 6.25 *Montrez que : $6n^3 + 2n^2 + 20n + 45$ est $O(n^3)$.*

Il est intéressant d'analyser comment la notation O interagit avec le logarithme et l'exposant. Une première remarque est qu'on peut négliger la base du logarithme et prendre toujours le logarithme en base 2. En effet, $log_b n = log_2 n / log_2 b$. En ce qui concerne l'exposant, on remarquera que la fonction 3^n n'est pas $O(2^n)$. Cependant elle est $O(2^{(cn)})$ en prenant par exemple $c = 2$. Pour cette raison, on introduit la notation $2^{O(f)}$. Par exemple, la notation $2^{O(n)}$ indique une fonction 2^{cn} pour une constante c. Ainsi 7^{45n} est $2^{O(n)}$. Notez cependant que 2^{n^2} n'est pas $2^{O(n)}$.

Définition 6.26 *Soit $g : \mathbf{N} \to \mathbf{N}$ une fonction sur les nombres naturels et M une MdT. On dit que M est $O(g)$ si la complexité en temps t de M est $O(g)$.*

Par exemple, dire qu'une machine M est $O(n)$ veut dire qu'ils existent $n_0, c \in \mathbf{N}$ tels que pour toute entrée w de taille $n \geq n_0$ le temps de calcul de M sur l'entrée w est au plus cn.

Exercice 6.27 *Montrez qu'il y a une MdT M qui décide le langage $L = \{w\sharp w \mid w \in \{0,1\}^*\}$ qui est $O(n^2)$.*

6.4 Variantes de MdT

Plusieurs variantes de MdT ont été considérées. Ces variantes n'affectent pas la notion de langage semi-décidable ou décidable mais peuvent changer de façon significative la complexité du calcul.

Machines multi-rubans Une MdT multi-rubans est une MdT qui dispose d'un nombre fini k de rubans. Sa définition formelle suit celle d'une MdT standard modulo le fait que le type de la fonction de transition δ est maintenant

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

Un pas de calcul se déroule de la façon suivante : en fonction de l'état courant et des symboles lus sur les k rubans, la machine va dans un autre état, remplace les symboles lus par d'autres symboles et déplace les têtes de lecture. Avec la directive S pour *stay* on a la possibilité de garder une tête de lecture à la même place.

Proposition 6.28 *Soit M une MdT multi-rubans. On peut construire une MdT standard M' qui simule M. Si la complexité de M est $t(n) \geq n$ la complexité de M' est $O(t(n)^2)$.*

Idée de la preuve. Supposons que la MdT M dispose de 3 rubans dont le contenu est $0101\sqcup^\omega$, $aab\sqcup^\omega$ et $ba\sqcup^\omega$ et dont les têtes de lecture sont en deuxième, troisième et première position respectivement. La MdT M' mémorise les trois rubans sur un seul ruban de la façon suivante :

$$\sharp 0\underline{1}01\sharp aa\underline{b}\sharp\underline{b}a\sharp\sqcup^\omega$$

On notera que M' dispose d'un nouveau symbole $\sharp$ pour séparer les rubans et que pour chaque symbole a de M on introduit un nouveau symbole $\underline{a}$. Le symbole souligné indique la position de la tête de lecture.

Un pas de calcul de M est simulé de la façon suivante :

- M' commence par parcourir son ruban de gauche à droite pour calculer les symboles en lecture et déterminer les actions à effectuer.
- Ensuite, M' effectue un deuxième passage dans lequel elle remplace le symbole en lecture (les symboles soulignés) par des nouveaux symboles et éventuellement déplace la tête de lecture (c'est-à-dire, remplace un symbole par un symbole souligné).
- Si le symbole souligné précède le symbole $\sharp$ et le calcul prévoit un déplacement à droite il est nécessaire d'allouer une nouvelle case. A cette fin, la machine M' décale à droite le contenu du ruban.

La borne $O(t(n)^2)$ sur le temps de calcul de la simulation est obtenue de la façon suivante. D'abord on observe que si la complexité de M est $t(n)$, la taille des rubans manipulés par M ne peut jamais dépasser $t(n)$. Ensuite on détermine le nombre d'opérations nécessaires à simuler un pas de calcul de M. Le premier passage est $O(t(n))$. Le deuxième passage est aussi $O(t(n))$ car le décalage à droite peut être effectué au plus k fois si la machine M comporte k rubans et chaque décalage peut être effectué en $O(t(n))$. •

Les machines multi-rubans permettent de donner une preuve simple du fait suivant.

Proposition 6.29 *Un langage L est décidable si et seulement si L et son complémentaire L^c sont semi-décidables.*

IDÉE DE LA PREUVE. ($\Rightarrow$) Par définition un langage décidable est semi-décidable. D'une MdT M qui décide L on obtient une MdT M' qui décide L^c simplement en échangeant les états finaux q_a et q_r.

($\Leftarrow$) Soient M et M' les MdT qui décident L et L^c, respectivement. On dérive une MdT N avec 2 rubans qui copie d'abord l'entrée w du premier au deuxième ruban et qui simule ensuite alternativement un pas de réduction de la machine M et un pas de réduction de la machine M'. La machine N accepte si M arrive à l'état q_a et elle refuse si M' arrive à l'état q'_a. La machine N termine toujours car tout mot w est accepté soit par M soit par M'. •

MdT non-déterministes Une MdT *non-déterministe* M est une MdT dont la fonction de transition δ a le type :

$$\delta : Q \times \Gamma \to 2^{(Q \times \Gamma \times \{L,R\})}$$

La notion de pas de calcul est adaptée immédiatement. Par exemple, on écrira

$$waqaw' \vdash_M wbq'w' \quad \text{si } (q', b, R) \in \delta(q, a)$$

Exercice 6.30 *Complétez la définition de pas de calcul d'une machine non-déterministe.*

La définition 6.3 de langage semi-décidable et décidable s'applique directement aux MdT non-déterministes.[6] On remarquera que pour qu'une entrée w soit acceptée il suffit qu'il existe un calcul qui mène de la configuration initiale à l'état q_a.

Proposition 6.31 *Soit N une MdT non-déterministe. On peut construire une MdT standard M qui simule N. Si la complexité de N est $t(n) \geq n$ la complexité de M est $2^{O(t(n))}$.*

IDÉE DE LA PREUVE. Dans une MdT non-déterministe N il y a une constante k qui borne le nombre d'alternatives possibles dans la suite du calcul. Ainsi on peut représenter le calcul d'une MdT non-déterministe comme un arbre éventuellement infini mais dont le branchement est borné par la constante k.

Les noeuds de cet arbre correspondent à des mots sur $\{0, \ldots, k-1\}^*$. On peut énumérer tous les noeuds de l'arbre en explorant l'arbre en largeur d'abord :

$$\epsilon, 0, \ldots, k-1, 00, \ldots, 0(k-1), 10, \ldots, 1(k-1), \ldots (k-1)0, \ldots, (k-1)(k-1), 000, \ldots$$

Une MdT peut calculer le successeur immédiat d'un mot π par rapport à cette énumération.

Pour simuler la machine N on utilise une MdT M avec 3 rubans. La proposition 6.28 nous assure qu'on peut toujours remplacer M par une MdT standard. Le premier ruban de M contient l'entrée w, le deuxième contient le chemin de l'arbre π qui est actuellement exploré et le troisième contient le ruban de la machine N lorsqu'elle calcule en effectuant les choix selon le chemin π.

Pour un chemin donné π, la machine M copie l'entrée du premier ruban au troisième et effectue ensuite un calcul en simulant l'exécution de N sur le chemin π.

– Le calcul peut bloquer car le chemin π ne correspond pas à un choix possible. Dans ce cas on considère le successeur immédiat de π et on itère.

[6] Ce n'est pas le cas pour la notion de fonction partielle récursive car il faut décider d'abord quel est le résultat d'une MdT non-déterministe...

– Le calcul arrive à la fin du chemin π mais la machine ne se trouve pas dans l'état q_a. Dans ce cas aussi on considère le successeur immédiat de π et on itère.
– Le calcul arrive à la fin du chemin π et la machine se trouve dans l'état q_a. Dans ce cas on accepte et on arrête le calcul.
– La simulation peut aussi noter qu'il ne reste plus de chemins à explorer et dans ce cas elle s'arrête et refuse.

Si la complexité de N est $t(n)$, la taille des chemins à considérer est aussi $O(t(n))$. Le nombre de chemins à simuler est $2^{O(t(n))}$. Donc la complexité de M est $2^{O(t(n))}$. Enfin, la MdT standard qui simule M est aussi $2^{O(t(n))}$ car $(2^{cn})^2$ est $2^{O(n)}$. •

Exercice 6.32 (1) *Montrez que les langages acceptés par un automate fini sont décidables.*

(2) *Montrez que la collection des langages décidables est stable par rapport aux opérations d'union, complémentaire, concaténation et itération.*

(3) *Montrez que la collection des langages semi-décidables est stable par rapport aux opérations d'union et concaténation.*

Suggestion : utilisez le non-déterminisme.

Thèse de Church-Turing Il est évident que le calcul d'une MdT est *effectif* dans le sens qu'une personne (une machine électronique) peut simuler le calcul d'une MdT à condition de disposer d'une quantité de papier (d'une quantité de mémoire) qui peut être étendue indéfiniment. La *thèse* de Church-Turing affirme que :

> Tout langage semi-décidable par une "procédure effective" est semi-décidable par une MdT.

On ne peut pas *démontrer* cette affirmation tant que la notion de "procédure effective" n'est pas formalisée. Le problème est qu'il n'y a pas de définition générale de "procédure effective". On dispose seulement d'*exemples* de "procédures effectives" (par exemple les MdT, les programmes assembleurs, les programmes Java, les systèmes de preuve,...) et ce qu'on peut faire est de démontrer que ces exemples sont *équivalents* au sens où ils permettent de semi-décider le même ensemble de langages. Nombreuses preuves de ce type ont été effectuées depuis les années 30 et ceci nous permet d'avoir un certain niveau de confiance dans la validité de la thèse.

6.5 Langages indécidables

On rappelle qu'il y a une bijection $\langle \_, \_ \rangle$ entre les mots finis et les couples de mots finis et que tout mot w peut être vu comme la représentation d'une MdT $\varphi(w)$. En particulier, on utilise la notation $M, M', \ldots$ pour des mots qui sont considérés comme des MdT. On écrit aussi $\varphi(M)(w)$ pour indiquer le résultat du calcul de la MdT représentée par $\varphi(M)$ sur une entrée w.

Définition 6.33 *Le langage H est défini par*

$$H = \{\langle M, w\rangle \mid \varphi(M)(w) \downarrow\}$$

Le langage H est semi-décidable par la MdT universelle. Le langage H formalise un problème intéressant qu'on appelle *problème de l'arrêt* : étant donné une MdT (un programme) M et une entrée w on se demande si le calcul de M sur l'entrée w termine.

On peut aussi considérer le comportement d'une machine M lorsque elle reçoit comme entrée le codage d'une machine M'. En particulier, on peut s'intéresser au résultat de l'application de la machine M à son propre codage.

Définition 6.34 *Le langage K est défini par*

$$K = \{M \mid \varphi(M)(M) \downarrow\}$$

On va montrer que les langages H et K *ne sont pas* décidables. Au passage, par la proposition 6.29 cela implique que les langages complémentaires H^c et K^c ne sont même pas semi-décidables.

Théorème 6.35 *Le langage K n'est pas décidable.*

Idée de la preuve. Si K est décidable il devrait y avoir une MdT $\varphi(M)$ telle que

$$\varphi(M)(M') \downarrow \text{ ssi } M' \in K^c$$

Si on applique $\varphi(M)$ à M on a deux possibilités :

1. Si $\varphi(M)(M) \downarrow$ alors $M \in K^c$ et donc $\neg\varphi(M)(M) \downarrow$.
2. Si $\neg\varphi(M)(M) \downarrow$ alors $M \notin K^c$ et donc $\varphi(M)(M) \downarrow$.

Les deux possibilités mènent à une contradiction, donc K^c n'est pas semi-décidable.[7] •

Plutôt que démontrer directement que H n'est pas décidable on va introduire une technique pour *réduire* l'analyse d'un langage à l'analyse d'un autre langage.

Définition 6.36 *Soient L, L' deux langages sur un alphabet Σ. On dit que L se réduit à L' et on écrit $L \leq L'$ s'il existe une fonction récursive $f : \Sigma^* \to \Sigma^*$ telle que*

$$w \in L \text{ ssi } f(w) \in L' \ .$$

Si $L \leq L'$ alors les méthodes de décision qu'on développe pour L' peuvent être appliquées à L aussi.

Proposition 6.37 *Si $L \leq L'$ et L' est semi-décidable (décidable) alors L est semi-décidable (décidable).*

Idée de la preuve. On sait qu'il existe une fonction récursive f telle que $w \in L$ ssi $f(w) \in L'$. Supposons que M_f soit une MdT qui calcule f et M' une MdT qui semi-décide L'. Pour semi-décider (décider) L il suffit de composer M' et M_f. •

Exemple 6.38 *On obtient que $K \leq H$ en utilisant la fonction $f(M) = \langle M, M\rangle$. Comme K n'est pas décidable, H ne peut pas être décidable non plus.*

[7] On appelle cette technique de preuve *diagonalisation*. On l'a déjà utilisée dans l'exercice 6.22.

Le fait que le problème de l'arrêt soit indécidable n'est que la pointe de l'iceberg...

Définition 6.39 *On dit que deux MdT sont extensionnellement équivalentes si elles terminent sur les mêmes entrées en donnant la même réponse (accepter/refuser).*[8]

Définition 6.40 *On dit qu'un langage $P \subseteq \Sigma^*$ est une propriété* extensionnelle *si P ne distingue pas les codages de deux machines qui sont extensionnellement équivalentes.*[9] *On dit aussi que P est triviale si P ou P^c est l'ensemble vide.*

Théorème 6.41 (Rice) *Toute propriété extensionnelle P non triviale est indécidable.*

IDÉE DE LA PREUVE. Soit $M_\emptyset$ le codage d'une MdT qui accepte le langage vide. Supposons que $M_\emptyset \notin P$ (autrement on montre que P^c est indécidable). Supposons aussi que $M_1 \in P$. Soit f la fonction qui associe au codage d'une MdT M le codage d'une MdT qui reçoit une entrée w, calcule $\varphi(M)(M)$ et si elle termine calcule $M_1(w)$. La machine $f(M)$ est extensionnellement équivalente à M_1 (et donc appartient à P) si et seulement si $M \in K$. Donc la fonction f montre que $K \leq P$. •

Exercice 6.42 *En utilisant le théorème de Rice, montrez que les langages suivants sont indécidables :*

(1) *L'ensemble K_ϵ des codages de MdT qui terminent sur l'entrée ϵ et acceptent ϵ.*

(2) *L'ensemble Tot des codages de MdT qui terminent sur toute entrée.*

(3) *L'ensemble Eq des codages de couples de MdT qui sont extensionnellement équivalentes.*

Une conséquence de (2) est qu'il ne peut pas y avoir un langage de programmation dans lequel on peut programmer exactement les fonctions totales. Il ne serait pas décidable de savoir si un programme de ce langage est bien formé. Il est donc nécessaire de donner des critères décidables qui assurent la terminaison mais qui excluent certains programmes qui terminent. Une conséquence de (3) est qu'on ne peut pas automatiser le problème de l'équivalence de deux programmes. Dans ce cas aussi on est amené à faire des approximations.

Exercice 6.43 *Montrez ou invalidez les assertions suivantes :*

1. *Il y a une MdT qui accepte les mots sur l'alphabet $\{0,1\}$ qui contiennent autant de 0 que de 1 (si la MdT existe, il suffira d'en donner une description informelle).*
2. *Rappel : si A et B sont deux langages, on écrit $A \leq B$ s'il existe une réduction de A à B.*
   Si A est sémi-décidable et $A \leq A^c$ alors A est décidable.
3. *L'ensemble des (codages de) MdT qui reconnaissent un langage fini est décidable.*

Exercice 6.44 *Montrez ou donnez un contre-exemple aux assertions suivantes :*

1. *L'ensemble des (codages de) MdT qui terminent sur le mot vide est décidable.*
2. *L'ensemble des (codages de) MdT qui divergent sur le mot vide est semi-décidable.*

[8] Il y a des variations possibles de cette définition. Par exemple, on peut dire que les machines sont équivalentes si elles calculent la même fonction partielle.

[9] En d'autres termes, si M et M' sont extensionnellement équivalentes alors soit $\{M, M'\} \subseteq P$ soit $\{M, M'\} \cap P = \emptyset$.

3. *L'ensemble des (codages de) MdT qui terminent sur le mot vide en* 10^{100} *pas de calcul est décidable.*

Exemple 6.45 *On termine en mentionnant (sans preuve) quelques problèmes indécidables remarquables.*

(1) *Soit* Σ *un alphabet et soit* $(v_1, w_1) \cdots (v_k, w_k)$ *une suite finie de couples de mots dans* Σ^*. *Le* problème de correspondance de Post *(PCP) consiste à déterminer s'ils existent* $n \geq 1$ *et* $i_1, \ldots, i_n \in \{1, \ldots, k\}$ *tels que :*

$$v_{i_1} \cdots v_{i_n} = w_{i_1} \cdots w_{i_n} \ .$$

Par exemple, considérez $\{(ab, a), (bcc, bb), (c, cc)\}$. *On ne peut pas concevoir un algorithme qui pour tout PCP décide si le problème a une solution. En d'autres termes, le problème de correspondance de Post est indécidable.*

(2) *Soit* $p(x_1, \ldots, x_n)$ *un polynôme de degré arbitraire avec variables* $x_1, \ldots, x_n$ *et avec coefficients dans* $\mathbf{Z}$. *Par exemple,* $p(x, y, z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$. *Le* dixième problème de Hilbert *consiste à déterminer si le polynôme* p *a des racines dans* $\mathbf{Z}$, *c'est-à-dire :*

$$\exists\, x_1, \ldots, x_n \in \mathbf{Z} \ \ p(x_1, \ldots, x_n) = 0$$

Ce problème a été proposé comme un challenge parmi d'autres en 1900 par D. Hilbert et il a été montré indécidable par Matijasevich en 1970 (le même problème sur les réels est décidable).

(3) *La logique du premier ordre est l'extension du calcul propositionnel où l'on introduit la quantification. Par exemple, on peut écrire* $\forall\, x \ \ \exists\, y \ \ A(x, y)$. *Une telle formule est valide si pour tout ensemble* $U \neq \emptyset$ *et pour toute relation binaire* R_A *sur* U *il est vrai que pour tout* $u \in U$ *il existe* $v \in V$ *tel que* $(u, v) \in R_A$. *La validité d'une formule du premier ordre est indécidable.*

(4) *On peut s'intéresser aux formules du premier ordre sur un alphabet particulier qui comprend les symboles* $+, *$ *et* $<$ *qui sont interprétés comme l'addition, la multiplication et l'inégalité de nombres naturels. De même, les quantificateurs sont interprétés maintenant sur les nombres naturels. Par exemple,* $\forall\, x \ \ \exists\, y \ \ x < y$ *est une formule qui dit que pour chaque nombre naturel* x *on peut trouver un nombre naturel* y *qui est strictement plus grand. La validité d'une formule du premier ordre (interprétée sur les nombres naturels) est (hautement) indécidable.*[10]

[10] On peut construire une hiérarchie qu'on appelle *hiérarchie arithmétique* de problèmes indécidables et toujours 'plus durs'.

7 Complexité : les classes P et NP

On s'intéresse au problèmes décidables en temps polynomial (déterministe ou non-déterministe).

Définition 7.1 *P (NP) est la classe des langages qui sont décidables par une MdT déterministe (non-déterministe) en temps $O(n^k)$ pour un certain k.*

Il suit de la définition que tout problème dans P est aussi dans NP. Les classes P et NP sont suffisamment robustes pour ne pas être affectées par une modification du modèle de calcul. Par exemple, ces classes ne dépendent pas du fait que les MdT disposent de un ou de plusieurs rubans. On peut même enrichir le modèle de calcul en supposant que la machine dispose d'une mémoire illimitée en accès direct (RAM pour *random access memory*). Dans une telle machine l'accès à une cellule de mémoire est effectué en $O(1)$. On peut démontrer qu'une MdT déterministe peut simuler une machine avec RAM avec une dégradation polynomiale des performances, c'est-à-dire qu'il y a un (petit) nombre k tel que si la machine avec RAM a complexité $O(t(n))$ la MdT qui la simule a complexité $O(t(n)^k)$.

Une grande partie des problèmes qui sont considérés dans un cours standard d'algorithmique font partie de la classe P. Par exemple, les problèmes de tri, la résolution de systèmes d'équations linéaires, les problèmes de recherche dans un arbre, le problème de la connectivité d'un graphe,... Dans la suite on va considérer un certain nombre de problèmes qui sont dans la classe NP.

Exemple 7.2 (1) *Le problème de savoir si une formule du calcul propositionnel est satisfiable est dans NP. Il suffit de deviner une affectation et de vérifier.*

(2) *Soit $G = (V, E)$ un graphe non-dirigé. Le problème du* circuit hamiltonien *consiste à déterminer s'il existe un parcours du graphe qui contient chaque sommet du graphe une et une seule fois. Un algorithme dans NP qui répond à la question devine une permutation des sommets et vérifie si elle correspond à un parcours dans le graphe.*

(2) *Soit V un ensemble de villes et d une fonction qui associe à chaque paire de villes (v, v') la distance $d(v, v') \geq 0$ pour aller de v à v'. Le problème du* voyageur de commerce[11] *est de déterminer s'il existe un parcours qui traverse chaque ville exactement une fois dont la longueur est inférieure à b. En d'autres termes, dans TSP on considère un graphe non-dirigé, complet (chaque couple de noeuds est connecté par une arête) et avec une fonction de coût sur les arêtes et on cherche à déterminer si le graphe contient un circuit hamiltonien dont le coût est inférieur à b. Un algorithme dans NP qui répond à la question devine une permutation des villes et vérifie si la somme des distances est inférieure à b.*[12]

7.1 Réduction polynomiale

Faute de pouvoir démontrer que les problèmes dans l'exemple 7.2 sont ou ne sont pas dans P, on va essayer de les comparer. A cette fin, on reprend la notion de *réduction* entre problèmes (définition 6.36) en ajoutant la contrainte que la réduction est calculable en temps polynomiale (déterministe).

[11] Aussi connu comme TSP pour *Travelling Salesman Problem*.

[12] Ce problème est aussi formulé comme un problème d'optimisation où l'on cherche à minimiser la longueur d'un parcours fermé.

Définition 7.3 *Soient L, L' deux langages sur un alphabet Σ. On dit que L se réduit à L' en* temps polynomial *et on écrit $L \leq_P L'$ s'il existe une fonction récursive $f : \Sigma^* \to \Sigma^*$ calculable en temps polynomial telle que*

$$w \in L \text{ ssi } w \in L'$$

Exemple 7.4 *Il y a une réduction polynomiale du problème du circuit hamiltonien au problème du voyageur de commerce. L'ensemble des noeuds correspond à l'ensemble des villes. La distance d est définie par :*

$$d(v, v') = \begin{cases} 1 & \text{si } (v, v') \text{ arête} \\ 2 & \text{autrement} \end{cases}$$

La constante b est égale au nombre des villes. Maintenant, on remarque :

- *S'il existe un parcours de longueur b alors ce parcours ne peut contenir que des chemins entre villes de longueur 1. Donc ce parcours correspond à un chemin hamiltonien.*
- *Inversement, s'il y a un chemin hamiltonien alors la réponse au problème du voyageur de commerce est positive.*

Exercice 7.5 *Une formule est en 3-CNF si elle est en CNF et chaque clause (disjonction de littéraux) comporte exactement 3 littéraux. Le problème 3-SAT consiste à déterminer si une formule en 3-CNF est satisfiable. Montrez que :*

(1) *3-SAT est dans NP.*

(2) *Une clause $\ell_1 \vee \cdots \vee \ell_n$ avec $n > 3$ peut être remplacée par*

$$(\ell_1 \vee \ell_2 \vee y_1) \wedge (\neg y_1 \vee \ell_3 \vee y_2) \wedge \cdots (\neg y_{n-3} \vee \ell_{n-1} \vee \ell_n)$$

où $y_1, \ldots, y_{n-3}$ sont des nouvelles variables.

(3) *Une clause avec 1 ou 2 littéraux peut être remplacée par une clause avec 3 littéraux.*

(4) *Conclure qu'il y a une réduction polynomiale de SAT à 3-SAT.*

Exercice 7.6 *Montrez que la notion de réduction polynomiale est transitive : $L_1 \leq_P L_2$ et $L_2 \leq_P L_3$ implique $L_1 \leq_P L_3$.*

7.2 SAT et NP-complétude

Définition 7.7 *Un problème L (langage) est NP-complet s'il est dans NP et si tout problème L' dans NP admet une réduction polynomiale à L.*

Dans un certain sens les problèmes NP-complets sont les plus durs. Si on trouve un algorithme polynomial pour un problème NP-complet alors on a un algorithme polynomial pour tous les problèmes de la classe NP. Un fait remarquable est que plusieurs problèmes naturels sont NP-complets.

Théorème 7.8 (Cook-Levin 1971) *Le problème SAT est NP-complet.*

Idée de la preuve. Soit L un langage décidé par une MdT M non déterministe polynomiale en temps $p(n)$. Donc $w \in L$ ssi à partir de la configuration initiale $q_0 w$ la machine M peut arriver à l'état q_a. On décrit une réduction polynomiale qui associe à chaque mot w une

formule en CNF A_w qui est satisfiable si et seulement si $w \in L$. L'idée est que la formule A_w va décrire les calculs possibles (M est non-déterministe !) de la machine M sur l'entrée w. La remarque fondamentale est qu'un calcul d'une machine de Turing en temps $p(n)$ sur un mot w de taille n peut être représenté par un tableau de taille $p(n) \times p(n)$ dont la case de coordonnées (i, j) contient la valeur du ruban au temps i et à la position j. Si le calcul termine avant $p(n)$ on peut toujours recopier le ruban jusqu'au temps $p(n)$.

On peut associer à chaque case (i, j) et à chaque symbole a une variable propositionnelle $x_{i,j,a}$ avec l'idée que $x_{i,j,a} = 1$ si et seulement si la case (i, j) contient le symbole a.

Ensuite on peut construire des formules (de taille polynomiale en n) qui assurent que :

- Exactement un symbole est dans chaque case.
- Les cases $(1, j)$ correspondent à la configuration initiale.
- Chaque case $(i + 1, j)$ est obtenue des cases $(i, j - 1), (i, j), (i, j + 1)$ selon les règles de la Machine.
- La configuration finale accepte.

Exemple 7.9 *On construit une CNF qui correspond au calcul de la MdT M dans l'exemple 6.4 sur l'entrée aab.*[13] *Le calcul de la MdT pourrait être :*

	1	2	3	4	5
1	q_0	a	a	b	$\sqcup$
2	a	q_0	a	b	$\sqcup$
3	a	a	q_0	b	$\sqcup$
4	a	a	b	q_1	$\sqcup$
5	a	a	b	$\sqcup$	q_a

Pour représenter le calcul on introduit les variables $x_{i,j,u}$ où $i, j \in \{1, \ldots, 5\}$ et $u \in \{a, b, q_0, q_1, q_a\}$. La configuration initiale est spécifiée par :

$$A_{init} = x_{1,1,q_0} \wedge x_{1,2,a} \wedge x_{1,3,a} \wedge x_{1,4,b} \wedge x_{1,5,\sqcup}$$

On doit imposer la contrainte que à chaque instant exactement un symbole est présent à chaque position. Par exemple, pour l'instant i à la position j on écrira :

$$A_{i,j} = (x_{i,j,a} \vee \cdots \vee x_{i,j,q_a}) \wedge (\neg x_{i,j,a} \vee \neg x_{i,j,b}) \wedge \cdots \wedge (\neg x_{i,j,q_1} \vee \neg x_{i,j,q_a})$$

L'objectif est d'arriver à une configuration qui contient l'état q_a. Cela revient à demander :

$$A_{accept} = \vee_{1 \leq i,j \leq 5} x_{i,j,q_a}$$

Enfin on doit décrire les 'règles de calcul' de la machine M. Par exemple, on pourrait exprimer $\delta(q_0, a) = (q_0, a, R)$ par la conjonction de formules de la forme

$$(x_{i-1,j-1,q_0} \wedge x_{i-1,j,a}) \rightarrow (x_{i,j-1,a} \wedge x_{i,j,q_0})$$

Il est possible de procéder d'une façon plus systématique. Une propriété intéressante des MdT est qu'à chaque instant le calcul est localisé dans une région de taille bornée. Si $w_1 q w_2 \vdash_M w_1' q' w_2'$ la différence entre les deux configurations est localisée dans une région de taille 3 qui

[13] Il s'agit d'un cas très spécial car la MdT en question se comporte comme un automate fini déterministe. Cependant les idées se généralisent.

comprend l'état et les deux symboles contiguës. L'idée est alors de regarder toutes les fenêtres de largeur 3 *et de hauteur* 2 *dans le tableau qui représente le calcul (il y en a un nombre polynomial) et de s'assurer que le contenu de chaque fenêtre est conforme aux règles de la machines.*

La formule en question peut être exprimée en CNF. Par exemple, on pourrait avoir une formule de la forme :

$$((x_1 \wedge x_2) \rightarrow (y_1 \wedge y_2)) \vee ((x_1 \wedge x_2) \rightarrow (w_1 \wedge w_2)) \vee ((x_1 \wedge x_2) \rightarrow (z_1 \wedge z_2))$$

pour dire que si deux cases contiennent les symboles a_1, a_2 (variables x_1, x_2) alors deux autres cases contiennent ou bien les symboles b_1, b_2 (variables y_1, y_2) ou bien les symboles c_1, c_2 (variables w_1, w_2) ou bien les symboles d_1, d_2 (variables z_1, z_2).

Une telle formule peut se ré-écrire en CNF comme suit.

$$\begin{array}{l}(\neg x_1 \vee \neg x_2 \vee y_1 \vee w_1 \vee z_1) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_1 \vee w_1 \vee z_2) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_1 \vee w_2 \vee z_1) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_1 \vee w_2 \vee z_2) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_2 \vee w_1 \vee z_1) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_2 \vee w_1 \vee z_2) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_2 \vee w_2 \vee z_1) \wedge \\ (\neg x_1 \vee \neg x_2 \vee y_2 \vee w_2 \vee z_2)\end{array}$$

*La transformation est exponentielle dans le nombre de possibilités (*3 *dans notre cas), mais pour une MdT donnée, ce nombre est borné par une constante.*

Exercice 7.10 *On dispose d'un ensemble $P = \{1, \ldots, m\}$ de pigeons et d'un ensemble $N = \{1, \ldots, n\}$ de nids. Le* principe du nid de pigeon[14] *est le suivant :*

1. *Chaque pigeon a un nid.*
2. *Chaque nid a au plus un pigeon.*

Décrivez le principe par une formule du calcul propositionnel en CNF qui utilise comme formules atomiques $o_{i,j}$ pour $i = 1, \ldots, m$ et $j = 1, \ldots, n$ où la validité de $o_{i,j}$ représente le fait que le pigeon i occupe le nid j. La formule en question doit être satisfiable si et seulement si $m \leq n$. Quelle est la taille de la formule en fonction de m, n ?

Remarque *: si on prend $m = n+1$ on obtient une formule en CNF qui n'est pas satisfiable. Cette formule est utilisée souvent comme un test pour les méthodes de preuve (Davis-Putnam, résolution,...)*

Exercice 7.11 *On dispose d'une grille 4×4 qui se décompose en* 4 *sous-grilles 2×2. On dénote par le couple (i, j), où $i, j \in \{1, 2, 3, 4\}$, les coordonnées d'une case de la grille. Chaque case de la grille contient un ensemble de nombres naturels contenu dans $\{1, 2, 3, 4\}$. On introduit* 64 *variables propositionnelles $x_{i,j,k}$ pour $i, j, k \in \{1, 2, 3, 4\}$ avec l'interprétation suivante :*

$x_{i,j,k}$ est 'vrai' si et seulement si la case de coordonnées (i, j) contient le nombre k.

[14] Traduction approximative de *pigeon principle*.

Soit A une formule qui utilise les variables $x_{i,j,k}$ et P une propriété de la grille. On dit que A exprime *P si, dans l'interprétation ci-dessus, A est satisfiable si et seulement si P est vérifiée. Par exemple, par la formule $A = x_{1,1,2} \vee x_{1,1,3}$ on exprime la propriété que la case de coordonnées $(1,1)$ contient ou bien 2 ou bien 3.*

1. *Définissez des formules en forme normale conjonctive qui expriment les propriétés suivantes :*
   (a) *La case de coordonnées $(2,2)$ contient au moins un numéro compris entre 1 et 4.*
   (b) *On ne peut pas trouver deux cases sur la première ligne qui contiennent le numéro 4.*
   (c) *La case $(3,2)$ contient au plus un numéro.*
2. *Donnez une borne supérieure au nombre de littéraux contenus dans une formule en forme normale conjonctive qui exprime la propriété suivante : il n'y a pas deux cases sur la même ligne, sur la même colonne ou sur la même sous-grille 2×2 qui contiennent le même numéro. Expliquez votre calcul.*

Exercice 7.12 *Pour $n \geq 1$ on introduit n^2 variables propositionnelles $x_{i,j}$ avec $1 \leq i, j \leq n$.*

(1) *Construisez une formule A_n en forme normale conjonctive qui a la propriété suivante : une affectation v satisfait A_n exactement quand il existe une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ telle que $v(x_{i,j}) = 1$ si et seulement si $\pi(i) = j$. Par exemple, pour $n = 2$ il n'y a que deux affectations qui peuvent satisfaire A_2 à savoir soit $(v(x_{1,1}) = v(x_{2,2}) = 1$ et $v(x_{1,2}) = v(x_{2,1}) = 0)$ soit $(v(x_{1,1}) = v(x_{2,2}) = 0$ et $v(x_{1,2}) = v(x_{2,1}) = 1)$. Écrivez explicitement A_n pour $n = 3$ et ensuite donnez le schéma de la formule A_n pour un n arbitraire. Suggestion : une permutation sur un ensemble fini X est la même chose qu'une fonction injective sur X.*

(2) *Un graphe fini non-dirigé G est un couple (N, E) où $N = \{1, \ldots, n\}$, $n \geq 2$ est un ensemble qui représente les noeuds du graphe et E est un ensemble de sous-ensembles de N de cardinalité 2 qui représente les arêtes du graphe. On dit que G admet un* circuit hamiltonien *s'il existe une permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ telle que*

$$\{\pi(1), \pi(2)\} \in E, \ldots, \{\pi(n-1), \pi(n)\} \in E$$

Montrez que le problème de savoir si un graphe admet un circuit hamiltonien a une réduction polynomiale au problème de la satisfiabilité d'une formule en CNF du calcul propositionnel. Suggestion : on utilise la formule A_n de l'exercice 7.12 pour spécifier l'existence d'une permutation et on ajoute des variables $y_{i,j}$, $i, j = 1, \ldots, n$, $i \neq j$ pour spécifier les arêtes du graphe.

Exercice 7.13 *On dispose d'un échiquier (une matrice carrée) $n \times n$. Une reine qui occupe une position de l'échiquier peut attaquer toutes les positions sur la même ligne, la même colonne ou sur les diagonales inclinées de 45 dégrées. On cherche à placer r reines sur l'échiquier de façon à ce qu'elles ne puissent pas s'attaquer mutuellement. A cette fin, écrivez une formule en CNF qui est satisfiable si et seulement si le problème a une solution. On utilisera des formules atomiques $o_{i,j}$ pour $i = 1, \ldots, m$ et $j = 1, \ldots, n$ où la validité de $o_{i,j}$ représente le fait qu'une reine occupe la position (i, j).*

Remarque *: la formule obtenue est aussi un test intéressant pour les méthodes de preuve. Par exemple, pour $n = r = 4$ ou $n = r = 8$ le problème a une solution.*

Exercice 7.14 *Soit A une matrice et b un vecteur à coefficients dans $\mathbf{Z}$. Le problème de programmation linéaire entière (ILP pour* integer linear programming*) consiste à déterminer s'il existe un vecteur $\vec{x}$ à coefficients dans $\mathbf{N}^m$ tel que $A\vec{x} = \vec{b}$.*[15] *Ce problème est dans NP. On utilise des notions d'algèbre linéaire pour montrer que si le problème a une solution alors il en a une dont la taille est polynomiale dans la taille de la matrice A. Ensuite on peut appliquer la méthode standard qui consiste à deviner un vecteur $\vec{x}$ et à vérifier qu'il est une solution. A partir de ce fait, le but de l'exercice est de montrer que le problème est NP-complet par réduction du problème SAT. Il peut être utile de considérer d'abord les problèmes suivants.*

- *Montrez qu'en introduisant des variables auxiliaires on peut exprimer la satisfaction d'une contrainte d'inégalité comme un problème d'ILP.*
- *Montrez qu'on peut exprimer la contrainte $x \in \{0, 1\}$.*
- *Montrez qu'on peut exprimer la contrainte $x = \overline{y}$ où $x, y \in \{0, 1\}$, $\overline{0} = 1$ et $\overline{1} = 0$.*
- *Montrez comment coder la validité d'une clause (disjonction de littéraux).*

Exercice 7.15 *Soit G un graphe non-dirigé (cf. exercice 8.6). Un k-clique est un ensemble de k noeuds de G qui ont la propriété que chaque couple de noeuds est connectée par une arête.*

Le langage CLIQUE est composé de couples $\langle G, k\rangle$ où (i) G est le codage d'un graphe, (ii) k est un nombre naturel et (iii) G contient comme sous-graphe un k-clique.

Le langage 3-SAT est composé de formules en forme normale conjonctive où chaque clause contient 3 littéraux.

1. *Montrez que le langage CLIQUE est dans NP.*
2. *On souhaite construire une réduction polynomiale de 3-SAT à CLIQUE. Si la formule A contient k clauses alors le graphe associé G_A contient k groupes de noeuds où chaque groupe est composé de 3 noeuds et chaque noeud est étiqueté par un littéral. Par exemple, si la clause est $(x \vee \neg y \vee z)$ alors on aura un groupe de 3 noeuds étiquetés avec x, $\neg y$ et z.*

   (a) *Décrivez les arêtes de G_A de façon à ce que le graphe G_A contienne une k-clique si et seulement si la formule A est satisfiable et dessinez le graphe G_A dans le cas où*

   $$A = (x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee \neg y)$$

   (la formule en question comporte seulement deux littéraux par clause mais la construction du graphe G_A s'applique aussi bien à ce cas).

   (b) *Quelle conclusion peut-on tirer de la construction précédente ? Motivez votre réponse :*

      i. *Si 3-SAT est un problème polynomiale déterministe alors CLIQUE est un problème polynomiale déterministe.*

      ii. *CLIQUE est un problème NP-complet.*

Exercice 7.16 *Un graphe (non-dirigé) G est composé d'un ensemble fini non-vide de noeuds N et d'un ensemble A d'arêtes qui connectent les noeuds. Formellement, une arête est un ensemble $\{i, j\}$ de noeuds de cardinalité 2. On dit que deux noeuds sont adjacents s'il y a une arête qui les connecte.*

[15] Comme pour le problème du voyageur de commerce, le problème ILP est souvent formulé comme un problème d'*optimisation*. Par exemple, il s'agit de minimiser une fonction linéaire $\vec{c}^T\vec{x}$ sous les contraintes $A\vec{x} = \vec{b}$ et $\vec{x} \geq 0$.

Problème du coloriage *Étant donné un graphe $G = (N, A)$ et un nombre naturel $k \geq 2$ on détermine s'il existe une fonction $c : N \to \{1, \ldots, k\}$ telle que si i, j sont deux noeuds adjacents alors $c(i) \neq c(j)$.*[16]

Problème de l'emploi du temps *Étant donné (i) un ensemble d'étudiants $E = \{1, \ldots, n\}$ ($n \geq 2$), (ii) un ensemble de cours $C = \{1, \ldots, m\}$ ($m \geq 2$), (iii) un ensemble de plages horaires $P = \{1, \ldots, p\}$ ($p \geq 2$) et (iv) une relations binaire R telle que $(i, j) \in R$ si et seulement si l'étudiant i suit le cours j on détermine s'il existe une fonction emploi du temps $edt : C \to P$ telle que si un étudiant suit deux cours différents $j \neq j'$ alors $edt(j) \neq edt(j')$.*

Démontrez ou donnez un contre-exemple aux assertions suivantes :

1. *Le problème de l'emploi du temps se réduit au problème du coloriage.*
2. *Le problème de l'emploi du temps se réduit en temps polynomial au problème du coloriage.*
3. *Le problème du coloriage est dans NP.*

Remarque 7.17 (1) *On connaît un bon millier de problèmes NP-complets. Cependant, certains problèmes comme l'*isomorphisme de graphes *(cf. exemple 6.14) résistent à une classification. A l'état de nos connaissances, il est possible que le problème de l'isomorphisme de graphes soit ni NP-complet ni dans P.*

(2) *La question de savoir s'il y a un langage dans NP qui n'est pas dans P est ouverte depuis 1971. C'est un problème naturel de la théorie de la complexité et il est aussi le problème le plus médiatisé de l'informatique théorique.* [17]

(3) *Une autre façon de mesurer la complexité du calcul d'une MdT est de compter l'espace, c'est-à-dire le nombre de cellules du ruban qu'elle utilise. La classe PSPACE (NPSPACE) est la classe des problèmes qui peuvent être résolus par une MdT déterministe (non-déterministe) en utilisant un espace polynomial dans la taille de l'entrée. Il n'est pas très difficile de montrer que PSPACE=NPSPACE. On en déduit immédiatement que $P \subseteq NP \subseteq PSPACE$ mais on ne sait pas si une de ces inclusions est stricte.*

(4) *Nombreuses autres classes de complexité ont été introduites. Par exemple : LOGSPACE, la classe des problèmes qui peuvent être résolus en espace logarithmique ($LOGSPACE \subseteq P$) et EXPTIME, la classe des problèmes qui peuvent être résolus en temps exponentiel ($PSPACE \subseteq EXPTIME$).*

[16] On peut voir les valeurs $\{1, \ldots, k\}$ comme des couleurs qu'on affecte aux noeuds, d'où le nom du problème.

[17] Le problème *P vs. NP* est cité parmi les "7 problèmes mathématiques du troisième millénaire" par la *Clay Foundation* à coté de *l'hypothèse de Riemann*, *la conjecture de Poincaré*, la *résolution des équations de Navier-Stokes*,... La preuve de la conjecture de Poincaré a été annoncée récemment, il ne reste donc que 6 problèmes...

8 Preuves par induction

On s'intéresse d'abord aux *définitions inductives*. Dans une définition inductive on construit un ensemble 'inductif' par stratifications successives et on dispose d'un principe de récurrence ainsi que d'un ordre implicite. Les ensembles librement engendrés constituent un exemple remarquable d'ensemble inductif. Par ailleurs, on peut généraliser la notion d'ordre et arriver à la notion d'ensemble *bien fondé*. Les ensembles bien fondés admettent un principe d'induction qui généralise le principe de récurrence habituel.

8.1 Ensembles inductifs

Soit A un ensemble, $X \subseteq A$ un sous-ensemble, et $F = \{f_i : A^{n_i} \to A \mid i \in I\}$ un ensemble d'opérations sur A. A partir de (A, X, F) on voudrait définir *inductivement* un ensemble $Ind(A, X, F)$ comme le plus petit sous-ensemble de A qui contient X et qui est stable par rapport aux opérations dans F, c'est-à-dire pour tout $i \in I$ si $y_1, \ldots y_{n_i} \in Y$ alors $f_i(y_1, \ldots, y_{n_i}) \in Y$.

Exemple 8.1 *Soit* $\mathbf{Z}$ *l'ensemble des nombres entiers et suc et* $+$ *les opérations successeur et addition, respectivement. On pourrait définir :*

(1) *L'ensemble des nombres naturels comme le plus petit sous-ensemble de* $\mathbf{Z}$ *qui contient* $\{0\}$ *et qui est stable par rapport à l'opération de successeur.*

(2) *L'ensemble des nombres pairs positifs comme le plus petit sous-ensemble de* $\mathbf{Z}$ *qui contient* $\{0, 2\}$ *et qui est stable par rapport à l'opération d'addition.*

Il n'est pas si évident qu'une définition inductive définit bien un ensemble. Il faut d'abord s'assurer que *le plus petit ensemble* dont parle la définition *existe*. A partir de (A, X, F) on peut définir une fonction $\mathcal{F} : 2^A \to 2^A$ comme suit :

$$\mathcal{F}(Z) = X \cup \{f_i(z_1, \ldots, z_{n_i}) \mid i \in I, z_j \in Z, j = 1, \ldots, n_i\}$$

On remarque que la condition $\mathcal{F}(Z) \subseteq Z$ est satisfaite si et seulement si $X \subseteq Z$ et Z est stable par rapport aux opérations dans F. Maintenant considérons l'intersection de tous les ensembles $Z \subseteq A$ qui satisfont cette condition :

$$Y = \bigcap \{Z \subseteq A \mid \mathcal{F}(Z) \subseteq Z\} \ . \tag{5}$$

Proposition 8.2 *Le plus petit ensemble* $Ind(A, X, F)$ *existe et est égale à* Y.

Idée de la preuve. Par définition, si $\mathcal{F}(Z) \subseteq Z$ alors $Y \subseteq Z$. Pour s'assurer de l'existence du plus petit ensemble tel que... il reste à démontrer que $\mathcal{F}(Y) \subseteq Y$. D'abord on observe que $\mathcal{F}$ est *monotone*, c'est-à-dire :

$$X_1 \subseteq X_2 \ \Rightarrow \ \mathcal{F}(X_1) \subseteq \mathcal{F}(X_2)$$

Si $\mathcal{F}(Z) \subseteq Z$ par définition de Y on dérive que $Y \subseteq Z$ et par monotonie que $\mathcal{F}(Y) \subseteq \mathcal{F}(Z)$. Donc

$$\mathcal{F}(Y) \subseteq \bigcap \{\mathcal{F}(Z) \mid Z \subseteq A, \mathcal{F}(Z) \subseteq Z\} \subseteq \bigcap \{Z \mid Z \subseteq A, \mathcal{F}(Z) \subseteq Z\} = Y \ .$$

•

Si $\mathcal{F}(Z) = Z$ on dit que Z est un point fixe de $\mathcal{F}$.

Proposition 8.3 (itération) (1) *L'ensemble Y défini par l'équation (5) est le plus petit point fixe de $\mathcal{F}$.*

(2) *De plus on peut donner une définition itérative de Y. Si on définit,*

$$\mathcal{F}^0 = \emptyset \qquad \mathcal{F}^{n+1} = \mathcal{F}(\mathcal{F}^n) \qquad \mathcal{F}^\omega = \bigcup_{n \geq 0} \mathcal{F}^n$$

alors $\mathcal{F}^n \subseteq \mathcal{F}^{n+1}$ et $\mathcal{F}^\omega = Y$.

IDÉE DE LA PREUVE. (1) On sait $\mathcal{F}(Y) \subseteq Y$. Par monotonie, $\mathcal{F}(\mathcal{F}(Y)) \subseteq \mathcal{F}(Y)$. Par définition de Y, $Y \subseteq \mathcal{F}(Y)$.

(2) On observe, $\mathcal{F}^n \subseteq Y$ implique par monotonie $\mathcal{F}^{n+1} \subseteq \mathcal{F}(Y) \subseteq Y$. Donc $\mathcal{F}^\omega \subseteq Y$. Ensuite, on vérifie que $\mathcal{F}(\mathcal{F}^\omega) \subseteq \mathcal{F}^\omega$, ce qui implique $Y \subseteq \mathcal{F}^\omega$. •

Proposition 8.4 *Tout ensemble Y défini inductivement à partir de (A, X, F) admet le principe d'induction suivant :*

$$\frac{Z \subseteq Y, \quad \mathcal{F}(Z) \subseteq Z}{Z = Y} \tag{6}$$

Si on explicite le principe dans le cas des nombres naturels $\mathbf{N}$, on obtient le principe de récurrence habituel :

$$\frac{Z \subseteq \mathbf{N}, \quad 0 \in Z, \quad \forall n \ n \in Z \to n+1 \in Z}{Z = \mathbf{N}} \tag{7}$$

Exercice 8.5 (transitivité) *Soit R une relation binaire sur un ensemble. Sa clôture réflexive et transitive R^* est la plus petite relation qui contient la relation identité, la relation R et telle que si $(x, y), (y, z) \in R^*$ alors $(x, z) \in R^*$. Montrez que R^* peut être vu comme un ensemble défini inductivement.*

Exercice 8.6 *Un graphe non-dirigé G est composé d'un ensemble fini non-vide de noeuds N et d'un ensemble A d'arêtes qui connectent les noeuds. Formellement, une arête est un ensemble $\{i, j\}$ de noeuds de cardinalité 2. Le* degré *d'un noeud i dans un graphe est le nombre d'arêtes qui le contiennent. Par exemple, un noeud isolé a degré 0. Démontrez en utilisant le principe de récurrence l'assertion suivante :*

Chaque graphe avec au moins 2 noeuds contient 2 noeuds avec le même degré.

8.2 Treillis complets et points fixes

Un ordre partiel $(L, \leq)$ est un ensemble L équipé d'une relation réflexive, anti-symétrique et transitive. Soit $X \subseteq L$ (éventuellement vide). Un élément $y \in L$ est une *borne supérieure* pour X si $\forall x \in X \ \ x \leq y$. Un élément $y \in L$ est le *sup* de X s'il est la plus petite borne supérieure. De façon duale, on définit la notion de *borne inférieure* et de *inf*.

Définition 8.7 *Un* treillis complet *est un ordre partiel $(L, \leq)$ tel que tout sous-ensemble a un sup.*

Définition 8.8 *Une fonction monotone f sur un ordre partiel L est une fonction qui respecte l'ordre. On dit que x est point fixe de f si $f(x) = x$.*

Proposition 8.9 (Tarski) (1) *Les parties d'un ensemble ordonnées par inclusion forment un treillis complet.*

(2) *Tout sous-ensemble d'un treillis complet a un inf.*

(3) *Toute fonction monotone sur un treillis complet a un plus grand et un plus petit point fixe qui s'expriment respectivement par* $sup\{x \mid x \leq f(x)\}$ *et* $inf\{x \mid f(x) \leq x\}$.

IDÉE DE LA PREUVE. (1) Le sup est l'union et l'inf est l'intersection d'ensembles.

(2) Soit $X \subseteq L$ et $BI(X)$ l'ensemble des bornes inférieures de X. On considère $z = sup(BI(X))$ et on montre que $z = inf(X)$.

(3) On pose $z = sup\{x \mid x \leq f(x)\}$. Si $f(y) = y$ alors $y \leq z$. Donc il reste à montrer que z est un point fixe. On montre d'abord que $z \leq sup\{f(x) \mid x \leq f(x)\} \leq f(z)$. Ensuite par monotonie, $f(z) \leq f(f(z))$ et par définition de z on arrive à $f(z) \leq z$. •

Exercice 8.10 *Soit* $(\mathbf{N} \cup \{\infty\}, \leq)$ *l'ensemble des nombres naturels avec un élément maximum* ∞, $0 < 1 < 2 < \ldots < \infty$. *Montrez que toute fonction monotone* f *sur cet ordre admet un point fixe, c'est-à-dire un élément* x *tel que* $f(x) = x$.

8.3 Ensembles librement engendrés

Nous allons considérer une forme particulièrement importante de définition inductive. Soit L un ensemble de symboles $\ell, \ell', \ldots$ avec arité $ar(\ell) \in \mathbf{N}$. On peut définir un ensemble $T(L)$ par :

$$\begin{array}{ll} T(L)_0 & = \{\ell \in L \mid ar(\ell) = 0\} \\ T(L)_{n+1} & = T(L)_n \cup \{(\ell, t_1, \ldots, t_n) \mid \ell \in L, ar(\ell) = n, t_i \in T(L)_n, i = 1, \ldots, n\} \\ T(L) & = \bigcup_{n \geq 0} T(L)_n \end{array}$$

On peut voir les éléments de $T(L)$ comme des *arbres finis* ordonnés dont les noeuds sont étiquetés par des symboles dans L de façon compatible avec leur arité. Maintenant, on peut associer à chaque symbole $\ell \in L$ une fonction à $ar(\ell)$ arguments sur $T(L)$ qui est définie par :

$$\ell(t_1, \ldots, t_n) = (\ell, t_1, \ldots, t_n) \tag{8}$$

Supposons maintenant $X \subseteq \{\ell \in L \mid ar(\ell) = 0\}$ et $\Sigma \subseteq L$ avec $X \cap \Sigma = \emptyset$. On peut définir un ensemble inductif $Y = Ind(T(L), X, \Sigma)$ qui est composé d'arbres finis dans $T(L)$ qui utilisent uniquement les symboles dans $X \cup \Sigma$ comme étiquettes. On dit que l'ensemble Y est *librement engendré* à partir de X et Σ.

Exemple 8.11 *L'ensemble des formules du calcul propositionnel peut être vu comme librement engendré à partir d'un ensemble* V *de symboles de 'variables' d'arité* 0 *et de symboles 'fonctionnels'* $\Sigma = \{\neg, \wedge, \vee\}$ *où* $ar(\neg) = 1$ *et* $ar(\wedge) = ar(\vee) = 2$. *En d'autres termes,*

$$Form = Ind(T(L), V, \Sigma)$$

On peut donc formuler un principe d'induction pour les formules du calcul propositionnel qui s'énonce de la façon suivante :

$$\frac{F \subseteq Form \quad V \subseteq F \quad (A, B \in F \text{ implique } \neg A, A \wedge B, A \vee B \in F)}{F = Form}$$

et qui correspond à l'intuition que Form est le plus petit ensemble tel que...[18]

Exercice 8.12 *On considère l'ensemble de symboles fonctionnels*

$$\Sigma = \{\epsilon, a, b\}$$

où $ar(\epsilon) = 0$ *et* $ar(a) = ar(b) = 1$. *Calculez l'ensemble librement engendré associé à* Σ. *Cet ensemble est-il isomorphe à un ensemble déjà considéré dans le cours ?*

8.4 Ensembles bien fondés

Dans un ensemble inductif on peut définir le rang d'un élément comme

$$rang(y) = min\{n \mid y \in \mathcal{F}^n\}$$

Ainsi on peut voir un ensemble inductif comme un ensemble stratifié (ou ordonné) en niveaux $0, 1, 2, \ldots$ On peut imaginer des ensembles avec une structure d'ordre différente. Par exemple, considérons $\mathbf{N} \cup \{\infty\}$ avec $\infty > n$ si $n \in \mathbf{N}$. Clairement, le principe de récurrence (7) n'est pas valide dans cet ensemble car même si un ensemble Z contient 0 et est stable par successeur il n'est pas forcement égal à $\mathbf{N} \cup \{\infty\}$. On va donc considérer un principe d'induction plus général qui s'applique aussi à des structures comme $\mathbf{N} \cup \{\infty\}$.

Définition 8.13 (ensemble bien fondé) *Un ensemble bien fondé est un couple* $(W, >)$ *où (1)* W *est un ensemble, (2)* $> \subseteq W \times W$ *est une relation transitive et (3) il n'existe pas de séquence infinie* $w_0 > w_1 > w_2 > \cdots$ *dans* W.[19]

Exemple 8.14 *L'ensemble des nombres naturels avec l'ordre usuel est bien fondé. L'ensemble des nombres entiers ou l'ensemble des nombres rationnels positifs ne le sont pas. L'ensemble* $\mathbf{N} \cup \{\infty\}$ *est bien fondé.*

Exemple 8.15 *L'ensemble des formules du calcul propositionnel ordonnées selon leur taille est bien fondé.*

Exercice 8.16 *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* $k \leq min(n, m)$ $(x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, x_k < y_k)$ *Est-il vrai que* $<$ *est un ordre bien fondé ? Donner soit une preuve soit un contre-exemple.*

Si $x \in W$ on dénote par $\downarrow(x)$ l'ensemble $\{y \mid x > y\}$ des éléments strictement plus petits que x.

Définition 8.17 (principe d'induction) *Soit* $(W, >)$ *un ordre bien fondé et* $Z \subseteq W$. *Chaque ordre bien fondé admet le principe de raisonnement par induction suivant :*

$$\frac{\forall x(\downarrow(x) \subseteq Z \rightarrow x \in Z)}{Z = W} \tag{9}$$

[18] Si on est pédant on devrait écrire $(\neg, A)$, $(\wedge, A, B)$ et $(\vee, A, B)$.

[19] Il en suit que $>$ est un ordre strict, c'est-à-dire pour tout $w \in W$, $w \not> w$.

Par exemple, considérons $W = \mathbf{N} \cup \{\infty\}$. Maintenant on ne peut pas appliquer le principe à $Z = \mathbf{N}$ car $\downarrow(\infty) \subseteq \mathbf{N}$ mais $\infty \notin \mathbf{N}$. Il est instructif d'expliciter le principe quand W est l'ensemble des nombres naturels avec l'ordre standard $>$. Dans ce cas la condition $\downarrow(x) \subseteq Z$ s'exprime aussi par : $\forall y < x \;\; y \in Z$. Donc pour montrer que $Z = \mathbf{N}$ il suffit de montrer : $\forall x \;\; \forall y < x \;\; y \in Z \rightarrow x \in Z$ c'est-à-dire : pour tout nombre x, il faut montrer que le fait que les éléments plus petits que x sont dans Z implique que x est dans Z aussi. On peut reformuler cette condition par :

$$(ind_{\mathbf{N}}) \quad \frac{0 \in Z \qquad \forall x > 0 \;\; ((\forall y < x \;\; y \in Z) \rightarrow x \in Z)}{Z = \mathbf{N}}$$

La condition est alors très proche du principe de récurrence standard :

$$(rec_{\mathbf{N}}) \quad \frac{0 \in Z \qquad \forall x > 0 \;\; (x - 1 \in Z \rightarrow x \in Z)}{Z = \mathbf{N}}$$

En effet on peut montrer que les deux principes sont équivalents.

Le principe d'induction (9) et la notion de bonne fondation sont deux faces de la même médaille.

Théorème 8.18 *Soit $(W, >)$ un ordre. $(W, >)$ est bien fondé si et seulement si le principe d'induction (9) est valide.*

Idée de la preuve. ($\Rightarrow$) Supposons que le principe (9) ne soit pas valide. Donc il y a un ensemble Z tel que $\forall x(\downarrow(x) \subseteq Z \rightarrow x \in Z)$ mais $x_0 \notin Z$. Mais alors il doit exister $x_1 \in \downarrow(x_0)$ tel que $x_1 \notin Z$. Par le même argument on trouve $x_2 \in \downarrow(x_1)$ tel que $x_2 \notin Z$. Donc on trouve une séquence infinie $x_0 > x_1 > x_2 > \cdots$ dans W ce qui contredit l'hypothèse de bonne fondation.

($\Leftarrow$) Soit

$$Z = \{x \mid \text{ il n'y a pas de suite descendante infinie à partir de } x\}$$

L'ensemble Z satisfait la condition

$$\forall x(\downarrow(x) \subseteq Z \rightarrow x \in Z)$$

ainsi par le principe d'induction (9), $Z = W$ et donc W est bien fondé. •

9 Méthodes de terminaison

Un problème fondamental en informatique consiste à démontrer la terminaison d'un programme ou d'un système de réduction.

Définition 9.1 (système de réduction) *Un système de réduction est un couple $(A, \to)$ où A est un ensemble et $\to \subseteq A \times A$.*

Exemple 9.2 *Un automate fini $M = (\Sigma, Q, q_0, F, \delta)$ où $\delta : \Sigma \times Q \to 2^Q$, définit un système de réduction sur l'ensemble des configurations $(\Sigma^* \times Q\times)$ par $(w, q) \to (w', q')$ si $w = aw'$ et $q' \in \delta(q, a)$.*

Exemple 9.3 *Le comportement d'un programme peut être défini par un système de réduction. Par exemple, considérons un langage impératif élémentaire composé de :*[20]

$$\begin{array}{ll} \textit{Variables} & v ::= x \mid y \mid \cdots \\ \textit{Expressions} & e ::= v \mid \mathsf{t} \mid \mathsf{f} \mid \cdots \\ \textit{Programmes} & P ::= \textit{skip} \mid v := e \mid \textit{if } e \textit{ then } P \textit{ else } P \mid \textit{while } e \textit{ do } P \mid P; P \end{array}$$

Une mémoire μ est une fonction qui affecte à chaque variable une valeur (dans notre cas, $\mathsf{true}, \mathsf{false}, \ldots$). Maintenant, le comportement d'un programme peut être défini par un système de réduction sur les couples (P, μ). D'abord on définit la valeur $[\![e]\!]\mu$ d'une expression e dans une mémoire μ : [21]

$$[\![x]\!]\mu = \mu(x), \quad [\![\mathsf{t}]\!]\mu = \mathsf{true}, \quad [\![\mathsf{f}]\!]\mu = \mathsf{false}, \cdots$$

Ensuite, on donne des règles pour évaluer un couple (P, μ).

$$\begin{array}{lll} (x := e, \mu) & \to (\textit{skip}, \mu[[\![e]\!]\mu/x]) & \\ (\textit{if } e \textit{ then } P \textit{ else } Q, \mu) & \to (Q, \mu) & \textit{si } [\![e]\!]\mu = \mathsf{false} \\ (\textit{if } e \textit{ then } P \textit{ else } Q, \mu) & \to (P, \mu) & \textit{si } [\![e]\!]\mu = \mathsf{true} \\ (\textit{while } e \textit{ do } P, \mu) & \to (\textit{skip}, \mu) & \textit{si } [\![e]\!]\mu = \mathsf{false} \\ (\textit{while } e \textit{ do } P, \mu) & \to (P; (\textit{while } e \textit{ do } P), \mu) & \textit{si } [\![e]\!]\mu = \mathsf{true} \\ (\textit{skip}; P, \mu) & \to (P, \mu) & \\ (P; Q, \mu) & \to (P'; Q, \mu') & \textit{si } (P, \mu) \to (P', \mu') \end{array}$$

Remarque 9.4 *La définition de la relation $\to$ est aussi de nature inductive. Soit Prog l'ensemble des programmes et Mem l'ensemble des mémoires. La relation $\to$ est le plus petit ensemble contenu dans $(Prog \times Mem)^2$ qui contient les couples $((P, \mu), (P', \mu'))$ qui satisfont une des première 6 conditions (règles) et qui est stable par rapport à une famille de fonctions $\{f_Q \mid Q \in Prog\}$ définie par $f_Q((P, \mu), (P', \mu')) = ((P; Q, \mu), (P'; Q, \mu'))$.*

Définition 9.5 (terminaison) *Un système de réécriture $(X, \to)$ termine s'il n'existe pas de suite infinie $x_0 \to x_1 \to x_2 \to \cdots$.*

[20] On présente ici la grammaire selon la notation BNF (*Backus-Naur Form*). Comme dans le cas des formules du calcul propositionnel, on pourrait voir les programmes comme les éléments d'un ensemble librement engendré.

[21] On remarquera l'analogie avec l'interprétation d'une formule du calcul propositionnel par rapport à une affectation.

Soit $\rightarrow^+$ la *clôture transitive* de la relation $\rightarrow$, à savoir :

$$\rightarrow^1 = \rightarrow \qquad \rightarrow^{n+1} = (\rightarrow) \circ (\rightarrow^n) \qquad \rightarrow^+ = \bigcup_{n \geq 1} \rightarrow^n$$

Il y a une relation naturelle entre terminaison et ordres bien fondés.

Définition 9.6 (plongement monotone) *Soit $(X, \rightarrow)$ un système de réécriture et $(W, >)$ un ordre bien fondé. X admet un plongement monotone dans W s'il existe une fonction $\mu : X \rightarrow W$ telle que $x \rightarrow y$ implique $\mu(x) > \mu(y)$.*

Théorème 9.7 *Soit $\underline{X} = (X, \rightarrow)$ un système de réécriture. Les assertions suivantes sont équivalentes :*

(1) *$\underline{X}$ termine.*

(2) *$(X, \rightarrow^+)$ est bien fondé.*

(3) *$\underline{X}$ admet un plongement monotone dans un ordre bien fondé.*

Idée de la preuve. (1) $\Rightarrow$ (2) Si $(X, \rightarrow^+)$ n'est pas bien fondé, on a une séquence $x_0 \rightarrow^+ x_1 \rightarrow^+ x_2 \rightarrow^+ \cdots$. Donc $(X, \rightarrow)$ ne termine pas.

(2) $\Rightarrow$ (3) Il suffit de prendre comme ordre bien fondé $(X, \rightarrow^+)$ et comme plongement l'identité.

(3) $\Rightarrow$ (2) Si $(X, \rightarrow)$ ne termine pas on a une séquence $x_0 \rightarrow x_1 \rightarrow \cdots$ ce qui induit une séquence $\mu(x_0) > \mu(x_1) > \cdots$ dans l'ordre bien fondé. •

Définition 9.8 *Soit $(X, \rightarrow)$ un système de réécriture.*

(1) *L'ensemble des successeurs immédiats d'un élément $x \in X$ est défini par :*

$$suc(x) = \{y \mid x \rightarrow y\}$$

(2) *L'ensemble des successeurs d'un élément $x \in X$ est défini par :*

$$suc^+(x) = \{y \mid x \rightarrow^+ y\}$$

(3) *On dit que le système est à branchement fini si pour tout $x \in X$, $suc(x)$ est fini.*

Proposition 9.9 *Soit $\underline{X} = (X, \rightarrow)$ un système de réécriture à branchement fini.*

(1) *Si $\underline{X}$ termine alors pour tout x, $suc^+(x)$ est fini.*

(2) *$\underline{X}$ termine si et seulement si il admet un plongement dans $\mathbf{N}$.*

Idée de la preuve. (1) Si $suc^+(x_0)$ est infini alors il existe $x_1 \in suc(x_0)$ tel que $suc^+(x_1)$ est infini donc il existe $x_2 \in suc(x_1)$ tel que $suc^+(x_2)$ ... On obtient ainsi une séquence $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots$ qui contredit l'hypothèse que le système termine.

(2) ($\Leftarrow$) Par le théorème 9.7. ($\Rightarrow$) On peut définir $\mu(x) = \sharp suc^+(x)$. Alternativement, on peut définir $\mu(x)$ comme la longueur de la plus longue séquence de réduction qui commence avec x. Dans les deux cas on vérifie que si $x \rightarrow y$ alors $\mu(x) > \mu(y)$. •

Exercice 9.10 *On considère des programmes impératifs* while *dont les variables prennent comme valeurs des nombres naturels. Montrez que le programme suivant termine où l'on sait que le test* Φ *termine et n'a pas d'effet de bord (c'est-à-dire que l'évaluation du test n'affecte pas la valeur associée aux variables) :*

$$\begin{array}{l} \textit{while } u > l+1 \textit{ do} \\ (r := (u+l) \textit{ div } 2; \\ \textit{if } \Phi \textit{ then } u := r \textit{ else } l := r) \end{array}$$

9.1 Confluence, terminaison et forme normale

Soit $(X, \to)$ un système de réécriture. Soit $\overset{*}{\to}$ la clôture réflexive et transitive de $\to$, à savoir $\overset{*}{\to} = \to^+ \cup Id$ où Id est la relation identité.

Définition 9.11 *On dit que le système est* confluent *si pour tout* $x \in X$

$$x \overset{*}{\to} y_1 \textit{ et } x \overset{*}{\to} y_2 \textit{ implique } \exists z \ (y_1 \overset{*}{\to} z \textit{ et } y_2 \overset{*}{\to} z) \tag{10}$$

On dit que le système est localement confluent *si pour tout* $x \in X$

$$x \to y_1 \textit{ et } x \to y_2 \textit{ implique } \exists z \ \ y_1 \overset{*}{\to} z \textit{ et } y_2 \overset{*}{\to} z$$

On dit que y *est une* forme normale *de* x *si* $x \overset{*}{\to} y$ *et* $\neg\exists z \ \ y \to z$.

Proposition 9.12 (Newman) *Soit* $\underline{X} = (X, \to)$ *un système de réécriture.*

(1) *Si* $\underline{X}$ *est confluent alors il est localement confluent.*

(2) *Si* $\underline{X}$ *termine et est localement confluent alors il est confluent.*

(3) *Si* $\underline{X}$ *est confluent alors chaque élément a au plus une forme normale.*

(4) *Si* $\underline{X}$ *termine et est localement confluent alors chaque élément a exactement une forme normale.*

Idée de la preuve. (1) Par définition.

(2) On montre que tout élément x est confluent. Si $x \to x_1 \overset{*}{\to} y_1$ et $x \to x_2 \overset{*}{\to} y_2$, par confluence locale $x_1 \overset{*}{\to} y$ et $x_2 \overset{*}{\to} y$. Par hypothèse inductive x_1 et x_2 sont confluents et ceci implique que x est confluent aussi.

(3) Si $x \overset{*}{\to} y_1$ et $x \overset{*}{\to} y_2$ et y_1, y_2 sont des formes normales alors $y_1 = y_2$ car autrement on contredit la confluence.

(4) Par (2) si le système est localement confluent alors il est confluent et donc si $x \overset{*}{\to} x_1, x_2$ et x_1, x_2 sont des formes normales alors $x_1 = x_2$. •

Exercice 9.13 *Soit* $(\{a, b, c, d\}, \to)$ *un système de réécriture où* $\to = \{(c, a), (c, d), (d, c), (d, b)\}$. *Dire si : le système termine, est localement confluent, est confluent.*

9.2 Ordre lexicographique

Il y a des systèmes de réécriture qui ne sont pas à branchement fini et dont la terminaison ne peut pas être démontrée par un plongement dans $\mathbf{N}$.

Exemple 9.14 *On considère le système de réécriture* $(\mathbf{N} \times \mathbf{N}, \to)$ *où*

$$(i+1, j) \to (i, k) \qquad (i, j+1) \to (i, j) \quad \textit{pour } i, j, k \in \mathbf{N}$$

Ce système n'est pas à branchement fini à cause de la première règle et il n'admet pas de plongement monotone dans $\mathbf{N}$. *Si* $\mu : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ *est monotone, on devrait avoir :*

$$k = \mu(1,1) > \mu(0,k) > \mu(0,k-1) > \cdots > \mu(0,0)$$

mais on a seulement k *nombres naturels sous* k *alors que la chaîne qu'on vient de construire a longueur* $k+1$. *Cependant, on peut montrer sa terminaison par plongement dans un ordre lexicographique que nous allons construire dans la suite.*

Définition 9.15 *Soient* $(A, >_A)$ *et* $(B, >_B)$ *deux ordres stricts. L'ordre lexicographique* $>_l$ *sur* $A \times B$ *est défini par :*

$$(x, y) >_l (x', y') \textit{ si } x >_A x' \textit{ ou } (x = x' \textit{ et } y >_B y')$$

Théorème 9.16 (1) *L'ordre lexicographique de deux ordres stricts est encore un ordre strict.*
(2) *L'ordre lexicographique de deux ordres bien fondés est encore bien fondé.*

IDÉE DE LA PREUVE. (1) Il est immédiat que $(x, y) \not>_l (x, y)$. Supposons $(x_1, y_1) >_l (x_2, y_2)$ et $(x_2, y_2) >_l (x_3, y_3)$.

- Si $x_1 > x_2$ alors $x_1 > x_3$ donc $(x_1, y_1) >_l (x_3, y_3)$.
- Si $x_1 = x_2$ alors $y_1 > y_2$. Si $x_2 > x_3$ alors $(x_1, y_1) >_l (x_3, y_3)$. Si par contre $x_2 = x_3$ alors $y_2 > y_3$ et donc $(x_1, y_1) >_l (x_3, y_3)$.

(2) Par contradiction. Supposons $(x_0, y_0) >_l (x_1, y_1) >_l \cdots$ Ceci implique $x_0 \geq x_1 \geq \cdots$ Comme A est bien fondé il existe i tel que $x_j = x_i$ pour $j \geq i$. Mais ceci implique $y_i > y_{i+1} > y_{i+2} > \cdots$ ce qui est impossible si B est bien fondé. •

Exercice 9.17 (1) *Montrez que le système dans l'exemple 9.14 termine.*
(2) *Montrez que le système* $(\mathbf{N} \times \mathbf{N}, \to)$ *où*

$$(i, j+1) \to (i, j) \textit{ et } (i+1, j) \to (i, i)$$

termine.
(3) *Trouvez (s'il existe) un plongement monotone du système précédent dans* $(\mathbf{N}, >)$.
(4) *Utilisez le principe d'induction pour démontrer l'existence d'une fonction (récursive)* $a : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ *telle que :*

$$\begin{array}{ll} a(0, n) & = n+1 \\ a(m+1, 0) & = a(m, 1) \\ a(m+1, n+1) & = a(m, a(m+1, n)) \end{array}$$

(5) *Calculez à l'aide d'un programme autant de valeurs $a(n,n)$ que possible.*

(6) *On étend l'ordre lexicographique à un produit $A = A_1 \times \cdots \times A_n$, $n \geq 3$, d'ordres bien fondés $(A_i, >_i)$:*

$$(x_1, \ldots, x_n) > (y_1, \ldots, y_n) \text{ si } \exists\, k \leq n \ \ (x_1 = y_1, \ldots, x_{k-1} = y_{k-1} \text{ et } x_k >_k y_k)$$

Montrez que $(A, >)$ est bien fondé.

Exercice 9.18 *Soit $A = \{a, b, c\}^*$ l'ensemble des mots finis sur l'alphabet $\{a, b, c\}$. Soit $\rightarrow$ une relation binaire sur A^* telle que :*

$$\begin{array}{rcl} w \rightarrow w' & \textit{ssi} & (w = w_1 aaw_2 \text{ et } w' = w_1 bcw_2) \text{ ou} \\ & & (w = w_1 bbw_2 \text{ et } w' = w_1 acw_2) \text{ ou} \\ & & (w = w_1 cccw_2 \text{ et } w' = w_1 acw_2) \end{array}$$

Donc w se réduit à w' si w' est obtenu de w en remplaçant ou bien un sous-mot aa par bc, ou bien un sous-mot bb par ac ou bien un sous-mot ccc par ac. Construisez un plongement monotone *dans un ordre bien fondé qui montre la terminaison de ce système de réduction. Suggestion : on peut commencer par remarquer que la dernière règle diminue le nombre de caractères.*

Exercice 9.19 *Soit $A = \{0, 1\}^*$ l'ensemble des mots finis sur l'alphabet $\{0, 1\}$. Si w est un mot on dénote par $|w|$ sa longueur. Soit $\rightarrow$ une relation binaire sur A telle que :*

$$w \rightarrow w' \quad \textit{ssi} \quad \exists\, w_1, w_2 \in A \ \ (w = w_1 01 w_2 \text{ et } w' = w_1 100 w_2)$$

En d'autres termes, un pas de réécriture revient à remplacer un sous-mot 01 *par le mot* 100*.*

1. *Trouvez deux fonctions $f_i : \mathbf{N} \rightarrow \mathbf{N}$, $i = 0, 1$ sur les nombres naturels $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$ telles que :*

   A *Si $n > n'$ alors $f_i(n) > f_i(n')$ pour $i = 0, 1$.*

   B *Pour tout n, $f_1(f_0(n)) > f_0(f_0(f_1(n)))$.*

2. *On définit une fonction $\mu : A \rightarrow \mathbf{N}$ par*

   $$\mu(\epsilon) = 0, \qquad \mu(wi) = f_i(\mu(w)) \text{ pour } i = 0, 1$$

   Montrez que μ est un plongement monotone du système de réécriture $(A, \rightarrow)$ dans l'ordre bien fondé $(\mathbf{N}, >)$.

   Suggestion *Montrez par récurrence sur $|w_2|$ que $\mu(w_1 01 w_2) > \mu(w_1 100 w_2)$.*

3. *Supposons que : $w_0 \rightarrow w_1 \rightarrow \cdots \rightarrow w_n$ et $|w_0| = m$. En d'autres termes, il y a n pas de réduction à partir d'un mot de longueur m.*

   (a) *Utilisez le plongement monotone μ pour donner une borne supérieure à n en fonction de m.*

   (b) *Démontrez ou donnez un contre-exemple à l'assertion suivante : $n \leq 2m^3 + 4m + 3$.*

Exercice 9.20 (ordre produit) *Soient $(A_i, >_i)$ pour $i = 1, \ldots, n$ des ordres bien fondés. On définit une relation $>$ sur le produit cartésien $A_1 \times \cdots \times A_n$ par*

$$(a_1, \ldots, a_n) > (a'_1, \ldots, a'_n) \text{ si } a_i \geq_i a'_i, i = 1, \ldots, n \text{ et } \exists\, i \in \{1, \ldots, n\}\ \ a_i >_i a'_i$$

(1) *La relation $>$ est-elle un ordre bien fondé ?*

(2) *Comparez la relation $>$ à l'ordre lexicographique sur le produit défini dans l'exercice 9.17.*

Exercice 9.21 *Considérons les programmes while :*

$$\begin{array}{l} \textit{while } m \neq n \textit{ do} \\ \quad \textit{if } m > n \textit{ then } m := m - n \textit{ else } n := n - m \end{array}$$

$$\begin{array}{l} \textit{while } m \neq n \textit{ do} \\ \quad \textit{if } m > n \textit{ then } m := m - n \\ \quad \textit{else } h := m; m := n; n := h \end{array}$$

Dire si les programmes terminent quand les variables varient sur les nombres naturels positifs.

Exercice 9.22 *Soit $A = \{a, b\}^*$ l'ensemble des mots finis sur l'alphabet $\{a, b\}$. Soit $\rightarrow$ une relation binaire sur A telle que :*

$$w \rightarrow w' \text{ ssi } w = w_1 abw_2 \text{ et } w' = w_1 bbaw_2$$

Donc w se réduit à w' si w' est obtenu de w en remplaçant un sous-mot ab avec bba.

Montrez ou invalidez les assertions suivantes (il est conseillé de s'appuyer sur les résultats démontrés dans le cours) :

1. *Le système de réduction $(A, \rightarrow)$ est à branchement fini.*
2. *Le système termine.*
3. *Le système est localement confluent.*
4. *Le système est confluent.*

9.3 Lemme de König

Soit $\mathbf{N}$ l'ensemble des nombres naturels et $\mathbf{N}^*$ l'ensemble des mots finis de nombres naturels. On va représenter un arbre comme l'ensemble des chemins possibles dans l'arbre. Formellement, on dit qu'un *arbre* D est un sous-ensemble de $\mathbf{N}^*$ qui satisfait les conditions suivantes :

(1) Si $w \in D$ et w' est un préfixe de w (c'est-à-dire $\exists\, w''\ \ w = w'w''$) alors $w' \in D$.

(2) Si $wi \in D$ et $j < i$ alors $wj \in D$.

On peut ainsi représenter des arbres infinis avec un nombre dénombrable de noeuds et même des arbres avec des noeuds qui ont un nombre dénombrable de fils. On dit qu'un arbre est à *branchement fini* si chaque noeud a un nombre fini de fils.

Remarque 9.23 *Parfois il est commode d'ajouter des symboles L sur les noeuds d'un arbre. Dans ce cas on définit un* arbre étiqueté *comme une fonction partielle $t : \mathbf{N}^* \rightharpoonup L$ dont le domaine de définition est un arbre.*

Lemme 9.24 (König) *Tout arbre à branchement fini qui comporte un nombre infini de noeuds admet un chemin infini.*

IDÉE DE LA PREUVE. On utilise le même argument mentionné dans la preuve de la proposition 9.9(1). Soit $\pi = i_1 \cdots i_k \in D$ un chemin tel que le sous-arbre de racine π est infini. Comme D est à branchement fini il existe un i_{k+1} tel que $\pi i_{k+1} \in D$ et le sous-arbre de racine πi_{k+1} est infini. En continuant ainsi on peut construire un chemin infini dans D. •

9.4 Ordre sur les mots

On donne un premier exemple d'application du lemme de König. Soit $(\Sigma, >)$ un ensemble bien fondé. Soit Σ^* l'ensemble des *mots finis* sur Σ. On écrit $w \succ w'$ si on peut obtenir w' de w en remplaçant un caractère $a \in \Sigma$ par un mot w'' tel que tous les caractères dans w'' sont plus petits que a dans la relation $>$. Par exemple, si $a > b > c$ alors on obtient $aab \succ abcbcb$ en remplaçant le deuxième caractère a par $bcbc$.

(1) On remarque que $\succ$ n'est pas transitive. Par exemple, $aab(\succ)(\succ)bac$ mais $aab \not\succ bac$. Soit donc $\succ^+$ la clôture transitive de $\succ$.

(2) Ajoutons à l'alphabet Σ un caractère $\top$ qui domine tous les autres et un caractère $\bot$ qui est dominé par tous les autres. Maintenant, toute suite

$$w_0 \succ w_1 \succ w_2 \succ \cdots$$

peut être réécrite comme

$$\top \succ w_0 \succ w_1 \succ w_2 \succ \cdots$$

en ajoutant $\top$ au début de la suite. Aussi, en supposant l'ordre alphabétique sur Σ, la suite

$$bc \succ defc \succ defg \succ dhifg \succ difg$$

se réécrit en

$$\top \succ bc \succ defc \succ defg \succ dhifg \succ d\bot ifg\ .$$

Ainsi le caractère h qui est remplacé par le mot vide dans la première suite est remplacé par $\bot$ dans la deuxième.

(3) On remarquera que le développement se représente aisément par un arbre étiqueté. En effet on a ajouté le caractère $\top$ pour commencer le calcul à la racine d'un arbre et le caractère $\bot$ pour couvrir le cas où un caractère est remplacé par le mot vide.

A partir de cette représentation il est facile de voir que l'ordre $\succ^+$ est bien fondé. Il suffit de montrer que la relation $\succ$ n'admet pas de suite descendante infinie. Si une telle suite existait, il serait possible de lui associer un arbre comme on vient de le voir. Or cet arbre est à branchement fini car un caractère est remplacé par un nombre fini de caractères. Par König, l'arbre en question doit comporter un chemin infini. Mais dans ce chemin on devrait trouver une suite infinie de caractères $a > b > c > \cdots$ ce qui contredit l'hypothèse.

Exercice 9.25 *Soit $\mathbf{N}^*$ l'ensemble des mots finis sur les nombres naturels avec l'ordre habituel. Montrez la terminaison du système :*

$$u(i+1)v \to uiiv \text{ pour } u, v \in \mathbf{N}^*$$

9.5 Ordre sur les multi-ensembles

Un multi-ensemble est un ensemble dont les éléments peuvent apparaître avec une certaine multiplicité.

Définition 9.26 *Un multi-ensemble M sur un ensemble A est une fonction $M : A \to \mathbf{N}$. Un multi-ensemble est fini si $\{x \mid M(x) \neq 0\}$ est fini.*

On utilise la notation $\{\!|\_|\!\}$ pour les multi-ensembles. Par exemple, $\{\!|a, b, a|\!\}$ est le multi-ensemble composé de deux occurrences de a et une de b. Certaines notations, opérations et relations disponibles sur les ensembles peuvent s'adapter aux multi-ensembles. Par exemple :

$$\begin{array}{ll} x \in M & \text{si } M(x) > 0 \\ M \subseteq N & \text{si } \forall\, x \in A \;\; M(x) \leq N(x) \\ (M \cup N)(x) & = M(x) + N(x) \\ (M \backslash N)(x) & = M(x) \dot{-} N(x) \end{array}$$

où $n \dot{-} m = n - m$ si $n - m \geq 0$ et 0 autrement. Si A est un ensemble on dénote avec $\mathcal{M}(A)$ l'ensemble des multi-ensembles *finis* sur A.

Remarque 9.27 *Un multi-ensemble fini sur A peut être représenté par un mot fini sur A. Cette représentation est unique si l'on suppose que l'opération de concaténation est commutative.*

Définition 9.28 *Soient $(A, >)$ un ordre strict et $M, N \in \mathcal{M}(A)$. On écrit : $M >_{\mathcal{M}(A)} N$ s'ils existent $X, Y \in \mathcal{M}(A)$ tels que $X \neq \emptyset$, $X \subseteq M$, $N = (M \backslash X) \cup Y$ et $\forall\, y \in Y \;\; \exists\, x \in X \;\; x > y$.*

Par exemple, si $A = \mathbf{N}$ avec l'ordre usuel alors $\{\!|5, 3, 1, 1|\!\} >_{\mathcal{M}(\mathbf{N})} \{\!|4, 3, 3, 1|\!\}$. Pour vérifier l'inégalité on peut choisir $X = \{\!|5, 1|\!\}$ et $Y = \{\!|4, 3|\!\}$ mais on peut aussi choisir $X = \{\!|5, 3, 3, 1|\!\}$ et $Y = \{\!|4, 3, 3, 1|\!\}$.

Proposition 9.29 *Si $>$ est un ordre strict sur A alors $>_{\mathcal{M}(A)}$ est un ordre strict sur $\mathcal{M}(A)$.*

Idée de la preuve. Il faut vérifier $X \not>_{\mathcal{M}(A)} X$ et que $>_{\mathcal{M}(A)}$ est transitif. La deuxième propriété demande un peu de travail. Supposons que :

$$X \cup \{\!|x_1, \ldots, x_m|\!\} >_{\mathcal{M}(A)} X \cup \{\!|y_1, \ldots, y_n|\!\} = X' \cup \{\!|w_1, \ldots, w_p|\!\} >_{\mathcal{M}(A)} X' \cup \{\!|z_1, \ldots z_q|\!\}$$

On peut décomposer X en $X_1 \cup X_2$ et $\{\!|y_1, \ldots, y_n|\!\}$ en $Y_1 \cup Y_2$ pour que $X' = X_1 \cup Y_1$ et $\{\!|w_1, \ldots, w_p|\!\} = X_2 \cup Y_2$. Donc on obtient

$$X_1 \cup X_2 \cup \{\!|x_1, \ldots, x_m|\!\} >_{\mathcal{M}(A)} X_1 \cup X_2 \cup Y_1 \cup Y_2 >_{\mathcal{M}(A)} X_1 \cup Y_1 \cup \{\!|z_1, \ldots, z_q|\!\}$$

Maintenant il suffit de vérifier que chaque élément dans $Y_1 \cup \{\!|z_1, \ldots, z_q|\!\}$ est dominé par un élément dans $X_2 \cup \{\!|x_1, \ldots, x_m|\!\}$. •

Exercice 9.30 *On écrit $X >_1 Y$ si Y est obtenu de X en remplaçant un élément de X par un multi-ensemble d'éléments strictement plus petits. Montrez que : (1) la relation $>_1$ n'est pas transitive (même si $>$ est total) et (2) la clôture transitive de $>_1$ est égale à $>_{\mathcal{M}(A)}$.*

Proposition 9.31 *L'ordre $(A, >)$ est bien fondé si et seulement si l'ordre $(\mathcal{M}(A), >_{\mathcal{M}(A)})$ est bien fondé.*

IDÉE DE LA PREUVE. ($\Leftarrow$) Si $a_0 > a_1 > \cdots$ est une suite décroissante dans A alors $\{\!|a_0|\!\} >_{\mathcal{M}(A)} \{\!|a_1|\!\} >_{\mathcal{M}(A)} \ldots$ en est une dans $\mathcal{M}(A)$.

($\Rightarrow$) Il suffit de montrer que $>_1$ est bien fondé. L'argument est similaire à celui utilisé pour l'ordre sur les mots et il fait aussi appel au lemme de König. •

Exercice 9.32 *Soit $\mathbf{N}^*$ les mots finis de nombres naturels. Montrez la terminaison du système :*

$$u(i+1)v \to iviui \text{ pour } u, v \in \mathbf{N}^*$$

A TD : Calcul propositionnel 1 (méthode de Davis Putnam)

La méthode de Davis Putnam permet de décider si une formule en forme normale conjonctive est satisfiable. On représente une formule A en CNF comme un ensemble (éventuellement vide) de clauses $\{C_1, \ldots, C_n\}$ et une clause C comme un ensemble (éventuellement vide) de littéraux. Dans cette représentation, on définit la substitution $[b/x]A$ d'une valeur booléenne $b \in \{0, 1\}$ dans A comme suit :

$$[b/x]A = \{[b/x]C \mid C \in A \text{ et } [b/x]C \neq 1\}$$

$$[b/x]C = \begin{cases} 1 & \text{si } (b = 1 \text{ et } x \in C) \text{ ou } (b = 0 \text{ et } \neg x \in C) \\ C\backslash\{\ell\} & \text{si } (b = 1 \text{ et } \ell = \neg x \in C) \text{ ou } (b = 0 \text{ et } \ell = x \in C) \\ C & \text{autrement} \end{cases}$$

On définit une fonction DP qui agit récursivement sur une formule A en CNF dans la représentation décrite ci-dessus :

function $DP(A) =$ case	
(1) $A = \emptyset$:	true
(2) $\emptyset \in A$	false
(3) $\{x, \neg x\} \subseteq C \in A$:	$DP(A\backslash\{C\})$
(4) $\{x\} \in A$:	$DP([1/x]A)$
(5) $\{\neg x\} \in A$:	$DP([0/x]A)$
(6) *else* :	choisir x dans A;
	$DP([0/x]A)$ or $DP([1/x]A)$

Dans (1), nous avons une conjonction du vide qui par convention est équivalente à true. Dans (2), A contient une clause vide. La disjonction du vide étant équivalente à false, la formule A est aussi équivalente à false. Dans (3), une clause contient un littéral et sa négation et elle est donc équivalente à true. Dans (4) et (5), A contient une clause qui est constituée uniquement d'une variable ou de sa négation. Ceci permet de connaître la valeur de la variable dans toute affectation susceptible de satisfaire la formule. Dans (6), nous considérons les deux valeurs possibles d'une affectation d'une variable.

Exercice A.1 *Appliquez DP aux formules* $\{\{x, \neg y\}, \{\neg x, y\}\}$ *et* $\{\{x, y\}, \{\neg x, y\}, \{x, \neg y\}, \{\neg x, \neg y\}\}$.

Exercice A.2 (1) *Montrez que si A est une fonction en CNF alors la fonction DP termine.*
(2) *Montrez que $DP(A)$ retourne* true *(*false*) si et seulement si A est satisfiable (ne l'est pas).*

Exercice A.3 *Fait : toute formule peut être transformée en CNF. Expliquez comment utiliser la méthode de Davis-Putnam pour décider la validité d'une formule.*

Exercice A.4 *Modifiez la fonction DP pour que, si la formule A est satisfiable, elle retourne une affectation v qui satisfait A.*

Exercice* A.5 *Réfléchissez aux structures de données et aux opérations nécessaires à la mise en oeuvre de l'algorithme en Java.*

B TD : Calcul Propositionnel 2 (équivalence et définissabilité)

Exercice B.1 *Montrez les équivalences logiques :*

$$(A \vee \mathbf{0}) \equiv A, \quad (A \vee \mathbf{1}) \equiv \mathbf{1}, \quad (A \vee B) \equiv (B \vee A),$$
$$((A \vee B) \vee C) \equiv (A \vee (B \vee C)), \quad (A \vee A) \equiv A$$
$$(A \wedge \mathbf{0}) \equiv \mathbf{0}, \quad (A \wedge \mathbf{1}) \equiv A, \quad (A \wedge B) \equiv (B \wedge A),$$
$$((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C)), \quad (A \wedge A) \equiv A,$$
$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C), \quad (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C),$$
$$\neg\neg A \equiv A, \quad \neg(A \vee B) \equiv ((\neg A) \wedge (\neg B)), \quad \neg(A \wedge B) \equiv ((\neg A) \vee (\neg B)) \ .$$

Exercice B.2 (1) *Montrez l'équivalence logique :*

$$(A \wedge B) \vee (\neg A \wedge B) \equiv B \tag{11}$$

(2) *On peut appliquer cette équivalence logique pour simplifier une forme normale disjonctive. Par exemple, considérez la fonction $f(x, y, z)$ définie par le tableau de vérité :*

$x \backslash yz$	00	01	11	10
0	0	1	1	0
1	1	1	1	1

Calculez la forme normale disjonctive de f et essayez de la simplifier en utilisant l'équivalence logique (11).

(3) *La présentation du tableau de vérité n'est pas arbitraire... Proposez une méthode graphique pour calculer une forme normale disjonctive simplifiée.*

Exercice B.3 *Soit f une fonction sur les nombres naturels. Dire qu'un problème est décidé en $O(f)$, signifie qu'on dispose d'un algorithme A et de n_0, k nombres naturels tels que pour toute entrée dont la taille n est supérieure à n_0, le temps de calcul de A sur l'entrée en question est inférieure à $k \cdot f(n)$.*

(1) *Montrez que la satisfaction d'une formule en DNF et la validité d'une formule en CNF peuvent être décidées en $O(n)$.*

(2) *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 représentation en DNF ou CNF de cette fonction est en $O(2^n)$. Peut-on appliquer l'équivalence logique (11) pour simplifier la représentation ?*

Exercice B.4 (if-then-else) *La fonction ternaire ITE est définie par $ITE(1, x, y) = x$ et $ITE(0, x, y) = y$. Montrez que toute fonction $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 0$ s'exprime par composition de la fonction ITE et des (fonctions) constantes 0 et 1.*

Exercice B.5 *L'*or exclusif $\oplus$ *(xor) est défini par $A \oplus B \equiv (A \wedge \neg B) \vee (\neg A \wedge B)$. Montrez que :*

(1) *$\oplus$ est associatif et commutatif.*

(2) *$x \oplus 0 \equiv x$ et $x \oplus x \equiv 0$.*

(3) *Toute fonction booléenne $f : \mathbf{2}^n \to \mathbf{2}$ peut être représentée à partir de 1, $\wedge$ et $\oplus$.*

Exercice B.6 (nand,nor) *Les fonctions binaires NAND et NOR sont définies par* $NAND(x, y) = NOT(AND(x, y))$ *et* $NOR(x, y) = NOT(OR(x, y))$. *Montrez que toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 0$, *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 que parmi les 16 fonctions binaires possibles il n'y en a pas d'autres qui ont cette propriété.*

C TD : Calcul Propositionnel 3 (clauses de Horn et circuits combinatoires)

Exercice C.1 *Une clause de (Alfred) Horn est une clause (c'est-à-dire une disjonction de littéraux) qui contient au plus un littéral positif. Une formule de Horn est une formule en CNF dont les clauses sont des clauses de Horn.*

(1) *Montrez que toute formule de Horn est équivalente à la conjonction (éventuellement vide) de clauses de Horn de la forme :*

$$\begin{array}{ll}(1) & x\\ (2) & \neg x_1 \vee \cdots \vee \neg x_n\\ (3) & \neg x_1 \vee \cdots \vee \neg x_n \vee x_{n+1}\end{array}$$

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.

(2) *Montrez qu'une formule de Horn réduite qui ne contient pas de clauses de la forme (1) ou qui ne contient pas de clauses de la forme (2) est satisfiable.*

(3) *Donnez une méthode efficace (temps polynomial) pour déterminer si une formule de Horn est satisfiable.*

Exercice C.2 *Un décodeur est un circuit avec n entrées $x_{n-1}, \ldots, x_0$ et 2^n sorties $y_{2^n-1}, \ldots, y_0$ tel que*

$$y_i = 1 \text{ ssi } i = (x_{n-1} \cdots x_0)_2$$

Réalisez un tel circuit.

Exercice C.3 *Un additionneur est un circuit booléen avec $2n$ entrées $x_{n-1}, y_{n-1}, \ldots, x_0, y_0$ et $n+1$ sorties $r_n, s_{n-1}, \ldots, s_0$ tel que*

$$(x_{n-1} \cdots x_0)_2 + (y_{n-1} \cdots y_0)_2 = (r_n s_{n-1} \cdots s_0)_2$$

On peut réaliser un additionneur en utilisant l'algorithme standard qui propage la retenue de droite à gauche.

(1) *Réalisez un circuit A avec 3 entrées x, y, r et deux sorties s, r' tel que*

$$(r's)_2 = (x)_2 + (y)_2 + (r)_2$$

(2) *Expliquez comment inter-connecter n circuits A pour obtenir un additionneur sur n bits.*

(3) *Montrez que dans le circuit en question le nombre de portes et la longueur du chemin le plus long sont proportionnels à n.*

Exercice* C.4 *Le but de cet exercice est de réaliser un additionneur dont le nombre de portes est encore polynomiale en n mais dont la longueur du chemin le plus long est proportionnelle à $lg(n)$. Pour éviter que la retenue se propage à travers tout le circuit, l'idée est d'anticiper sa valeur. Ainsi pour additionner 2 vecteurs de longueur n, on additionne les premiers $n/2$ bits (ceux de poids faible) et en même temps on additionne les derniers $n/2$ bits (ceux de poids fort) deux fois (en parallèle) une fois avec retenue initiale 0 et une fois avec retenue initiale 1. On applique cette méthode récursivement sur les sous-vecteurs de longueur $n/4, n/8, \ldots$ selon le principe* diviser pour régner*.*

(1) *Construisez explicitement un tel circuit pour $n = 4$.*

(2) *Déterminez en fonction de n le nombre de portes et la longueur du chemin le plus long du circuit obtenu.*

D TD : Système de preuve de Gentzen et compacité

Rappel : voici le système de preuve de Gentzen.

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

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

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

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

Exercice D.1 *Montrez que :*

(1) *Un séquent $A, \Gamma \vdash A, \Delta$ est valide.*

(2) *Pour chaque règle d'inférence la conclusion est valide si et seulement si les hypothèses sont valides.*

Exercice D.2 (sous-formule) *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* D.3 (affaiblissement) *Montrez que si le séquent $\Gamma \vdash \Delta$ est dérivable alors le séquent $\Gamma \vdash A, \Delta$ l'est aussi.*

Exercice D.4 (implication) *Dans le système de Gentzen on peut donner un traitement direct de l'implication :*

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

Démontrez la correction et complétude du système de Gentzen étendu avec ces règles.

Exercice D.5 *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 $A \rightarrow B \equiv \neg A \vee B$.*

Exercice D.6 (coupure) *La règle de* coupure *(ou* cut*) est :*

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

Montrez que le système de Gentzen étendu avec cette règle est toujours correct (et complet).

Exercice D.7 *Soit T un ensemble de formules. On écrit $T \models A$ si pour toute affectation v, si v satisfait T alors v satisfait A. Montrez que si $T \models A$ alors il existe T_0 sous-ensemble fini de T tel que $T_0 \models A$. Suggestion : utilisez le théorème de compacité.*

E TD : Résolution

Exercice E.1 *Montrez que la règle d'inférence suivante est valide :*

$$\frac{A \vee \neg C \qquad B \vee C}{A \vee B} \tag{12}$$

Exercice E.2 *Pour représenter les formules en CNF on adopte la même notation ensembliste utilisée pour décrire la méthode de Davis-Putnam.*

- *Une clause C est un ensemble de littéraux.*
- *Une formule A est un ensemble de clauses.*

Nous considérons la règle :

$$\frac{A \cup \{C \cup \{x\}\} \cup \{C' \cup \{\neg x\}\} \quad x \notin C \quad \neg x \notin C'}{A \cup \{C \cup \{x\}\} \cup \{C' \cup \{\neg x\}\} \cup \{C \cup C'\}} \tag{13}$$

Dans la suite on appelle (13) règle de résolution.[22] *L'effet de l'application de la règle consiste à ajouter une nouvelle clause $C \cup C'$ qu'on appelle* résolvant *des deux clauses $C \cup \{x\}$ et $C' \cup \{\neg x\}$.*

(1) *Montrez que l'hypothèse est logiquement équivalente à la conclusion.*

(2) *Conclure que si la conclusion n'est pas satisfiable alors l'hypothèse n'est pas satisfiable. En particulier, si la conclusion contient la clause vide alors l'hypothèse n'est pas satisfiable.*

Fait Si une formule A en CNF n'est pas satisfiable 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. La méthode peut être implémentée itérativement. A chaque itération on ajoute toutes les clauses qui sont un résolvant de deux clauses. Cette itération termine forcement car le nombre de clauses qu'on peut construire est fini. Parfois, il convient de représenter la dérivation comme un graphe dirigé acyclique (ou DAG pour *directed acyclic graph*) dont les noeuds sont étiquetés par les clauses. Initialement on a autant de noeuds que de clauses et pas d'arêtes. Chaque fois qu'on applique la règle de résolution (13) on introduit un nouveau noeud qui est étiqueté avec la clause résolvant $C \cup C'$ et deux nouvelles arêtes qui vont des noeuds étiquetés avec les clauses $C \cup \{x\}$ et $C' \cup \{\neg x\}$ vers le noeud étiqueté avec la clause $C \cup C'$.

Exercice E.3 *Soit A une formule en CNF et C une clause. Expliquez comment utiliser la méthode de résolution pour établir si l'implication $A \to C$ est valide.*

Exercice E.4 *Construire la formule A en CNF qui correspond au principe du nid de pigeon avec 2 pigeons et 1 nid. Appliquez la règle de résolution. Même problème avec 2 pigeons et 2 nids.*

Exercice E.5 *Soit A une formule en CNF avec m variables et n clauses. Montrez qu'il y a au plus $m \cdot (n \cdot (n-1)/2)$ façons d'appliquer la règle de résolution.*

Exercice E.6 *Un exercice de révision. On considère les formules en CNF suivantes :*

[22] Sans les conditions $x \notin C$ et $\neg x \notin C'$ on pourrait par exemple 'simplifier' les clauses $\{x\}$ et $\{\neg x\}$ en $\{x, \neg x\}$.

1. $\neg x \vee (\neg y \vee x)$
2. $(x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (x \vee \neg y) \wedge \neg x$.
3. $(x \vee y) \wedge (z \vee w) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg w)$.

Pour chaque formule :

1. *Si la formule est valide calculez une preuve de la formule dans le système de Gentzen.*
2. *Si la formule est satisfiable mais pas valide calculez une affectation qui satisfait la formule en utilisant la méthode de Davis-Putnam.*
3. *Si la formule n'est pas satisfiable dérivez la clause vide en utilisant la méthode par résolution.*

F TD : Langages formels et automates finis

Exercice F.1 *Montrez que pour tout langage L, $L^* = (L^*)^*$.*

Exercice F.2 *Montrez qu'il existe des langages L_1 et L_2 tels que $(L_1 \cup L_2)^* \neq L_1^* \cup L_2^*$.*

Exercice F.3 *Montrez qu'il existe des langages L_1 et L_2 tels que $(L_1 \cdot L_2)^* \neq L_1^* \cdot L_2^*$.*

Exercice F.4 *Pour chacun des langages suivants, construire un automate fini non déterministe qui l'accepte :*

1. *Les représentations binaires des nombres pairs.*
2. *Le langage des mots sur l'alphabet $\{a, b\}$ contenant ou bien la chaîne aab ou bien la chaîne aaab.*
3. *Le langage des mots sur l'alphabet $\{0, 1\}$ dont le troisième caractère de droite existe et est égale à 1.*

Construire des automates déterministes pour les langages décrits ci-dessus.

Exercice* F.5 *Soient M un AFD qui accepte un langage L et N_1, N_2 deux AFN qui acceptent les langages L_1, L_2, respectivement (sur un alphabet Σ fixé).*

1. *Montrez qu'on peut construire un AFD qui accepte le langage complémentaire $\Sigma^* \backslash L$.*
2. *Montrez qu'on peut construire un AFN qui accepte le langage $L_1 \cup L_2$ et le langage itéré $(L_1)^*$.*
3. *Conclure que la classe des langages acceptés par un AFD est stable par union, intersection, complémentaire et itération.*

G TD : Calculabilité 1 (machines de Turing et énumérations)

Exercice G.1 *Donnez la description formelle d'une MdT qui décide le langage* $\{w\sharp w \mid w \in \{0,1\}^*\}$.

Exercice G.2 *Donnez la description formelle d'une MdT qui décide le langage des mots sur l'alphabet* $\{0\}$ *dont la longueur est une puissance de* 2 *:* 2^0, 2^1, $2^2, \ldots$

Exercice G.3 *Décrivez informellement une MdT qui décide le langage :*

$$\{a^i b^j c^k \mid i \cdot j = k \text{ et } i, j, k \geq 1\} \ .$$

Exercice G.4 *Soit* $\Sigma = \{0,1\}$ *et* $suc : \Sigma^* \to \Sigma^*$ *la fonction 'successeur' en base* 2 *telle que :*

$$(suc(w))_2 = (w)_2 + 1$$

Montrez que suc est récursive.

Exercice G.5 *On peut énumérer les couples de nombres naturels en procédant 'par diagonales' :*

$$(0,0), \quad (1,0),(0,1), \quad (2,0),(1,1),(0,2), \quad (3,0)\ldots$$

Montrez que la fonction $\langle m,n\rangle = (m+n)(m+n+1)/2 + n$ *est une bijection entre* $\mathbf{N}\times\mathbf{N}$ *et* $\mathbf{N}$. *Décrire un algorithme pour calculer la fonction inverse.*

Exercice G.6 *On définit les fonctions* $\langle\_\rangle_k : \mathbf{N}^k \to \mathbf{N}$ *pour* $k \geq 2$ *:*

$$\begin{array}{ll} \langle m,n\rangle_2 & = \langle m,n\rangle \\ \langle n_1,\ldots,n_k\rangle_k & = \langle\langle n_1,\ldots,n_{k-1}\rangle_{k-1}, n_k\rangle \text{ si } k \geq 3 \end{array}$$

Montrez que les fonctions $\langle\_\rangle_k$ *sont des bijections.*

H TD : Calculabilité 2 (énumérations et indécidabilité)

Exercice H.1 *On considère l'ensemble* $\mathbf{N}^*$ *des mots finis de nombres naturels. Notez que* $\mathbf{N}^*$ *est en correspondance bijective avec* $\bigcup_{k\geq 0} \mathbf{N}^k$*. Définissez une bijection entre* $\mathbf{N}^*$ *et* $\mathbf{N}$*.*

Exercice H.2 *Soit* $\Sigma = \{a, b, \ldots, z\}$ *un alphabet fini. On peut énumérer les éléments de* Σ^* *comme suit :*

$$\epsilon, \quad a, b, \ldots, z, \quad aa, \ldots, az, ba, \ldots, bz, za, \ldots, zz, \quad aaa, \ldots$$

Si Σ *contient* k *éléments on aura* k^0 *mots de longueur* 0*,* k *mots de longueur* 1*,* k^2 *mots de longueur* 2*, ... Définissez une bijection entre* Σ^* *et* $\mathbf{N}$*.*

Exercice* H.3 (1) *Montrez qu'un langage est semi-décidable si et seulement si il est le domaine de définition d'une fonction partielle récursive.*

(2) *On dit qu'un langage* $L \subseteq \Sigma^*$ *est* récursivement énumérable *s'il est l'image d'une fonction partielle récursive. Montrez qu'un langage* L *est récursivement énumérable si et seulement si il est semi-décidable.*

Suggestion : Soit M *une MdT et* $w_0, w_1, w_2, \ldots$ *une suite d'entrées. On peut simuler* M *sur* w_0 *pour* 0 *pas, sur* w_0 *pour* 1 *pas, sur* w_1 *pour* 0 *pas, sur* w_0 *pour* 2 *pas, sur* w_1 *pour* 1 *pas, sur* w_2 *pour* 0 *pas,...*

Exercice* H.4 (1) *Montrez que les langages acceptés par un AFN sont décidables.*

(2) *Montrez que la collection des langages décidables est stable par rapport aux opérations d'union, complémentaire, concaténation et itération.*

(3) *Montrez que la collection des langages semi-décidables est stable par rapport aux opérations d'union et concaténation.*

Suggestion : utilisez le non-déterminisme.

Exercice H.5 *Montrez ou invalidez les assertions suivantes :*

1. *Il y a une MdT qui accepte les mots sur l'alphabet* $\{0, 1\}$ *qui contiennent autant de* 0 *que de* 1 *(si la MdT existe, il suffira d'en donner une description informelle).*
2. *Rappel : si* A *et* B *sont deux langages, on écrit* $A \leq B$ *s'il existe une réduction de* A *à* B*.*

   Si A *est semi-décidable et* $A \leq A^c$ *alors* A *est décidable.*
3. *L'ensemble des (codages de) MdT qui reconnaissent un langage fini est décidable.*

Exercice H.6 *Montrez ou donnez un contre-exemple aux assertions suivantes :*

1. *L'ensemble des (codages de) MdT qui terminent sur le mot vide est décidable.*
2. *L'ensemble des (codages de) MdT qui divergent sur le mot vide est semi-décidable.*
3. *L'ensemble des (codages de) MdT qui terminent sur le mot vide en* 10^{100} *pas de calcul est décidable.*

I TD : Complexité (réductions polynomiales)

Exercice I.1 *Un graphe (non-dirigé) G est composé d'un ensemble fini non-vide de noeuds N et d'un ensemble A d'arêtes qui connectent les noeuds. Formellement, une arête est un ensemble $\{i, j\}$ de noeuds de cardinalité 2. On dit que deux noeuds sont adjacents s'il y a une arête qui les connecte.*

Problème du coloriage *Étant donné un graphe $G = (N, A)$ et un nombre naturel $k \geq 2$ on détermine s'il existe une fonction $c : N \to \{1, \ldots, k\}$ telle que si i, j sont deux noeuds adjacents alors $c(i) \neq c(j)$.*[23]

Problème de l'emploi du temps *Étant donné (i) un ensemble d'étudiants $E = \{1, \ldots, n\}$ ($n \geq 2$), (ii) un ensemble de cours $C = \{1, \ldots, m\}$ ($m \geq 2$), (iii) un ensemble de plages horaires $P = \{1, \ldots, p\}$ ($p \geq 2$) et (iv) une relations binaire R telle que $(i, j) \in R$ si et seulement si l'étudiant i suit le cours j on détermine s'il existe une fonction emploi du temps $edt : C \to P$ telle que si un étudiant suit deux cours différents $j \neq j'$ alors $edt(j) \neq edt(j')$.*

Démontrez ou donnez un contre-exemple aux assertions suivantes :

1. *Le problème de l'emploi du temps se réduit au problème du coloriage.*
2. *Le problème de l'emploi du temps se réduit en temps polynomial au problème du coloriage.*
3. *Le problème du coloriage est dans NP.*

Exercice I.2 *Soit G un graphe non-dirigé (cf. exercice I.1). Un k-clique est un ensemble de k noeuds de G qui ont la propriété que chaque couple de noeuds est connectée par une arête.*

Le langage CLIQUE est composé de couples $\langle G, k\rangle$ où (i) G est le codage d'un graphe, (ii) k est un nombre naturel et (iii) G contient comme sous-graphe un k-clique.

Le langage 3-SAT est composé de formules en forme normale conjonctive où chaque clause contient 3 littéraux.

1. *Montrez que le langage CLIQUE est dans NP.*
2. *On souhaite construire une réduction polynomiale de 3-SAT à CLIQUE. Si la formule A contient k clauses alors le graphe associé G_A contient k groupes de noeuds où chaque groupe est composé de 3 noeuds et chaque noeud est étiqueté par un littéral. Par exemple, si la clause est $(x \vee \neg y \vee z)$ alors on aura un groupe de 3 noeuds étiquetés avec x, $\neg y$ et z.*

   (a) *Décrivez les arêtes de G_A de façon à ce que le graphe G_A contienne une k-clique si et seulement si la formule A est satisfiable et dessinez le graphe G_A dans le cas où*

   $$A = (x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee \neg y)$$

   (la formule en question comporte seulement deux littéraux par clause mais la construction du graphe G_A s'applique aussi bien à ce cas).

   (b) *Quelle conclusion peut-on tirer de la construction précédente ? Motivez votre réponse :*

[23] On peut voir les valeurs $\{1, \ldots, k\}$ comme des couleurs qu'on affecte aux noeuds, d'où le nom du problème.

i. Si 3-SAT est un problème polynomiale déterministe alors CLIQUE est un problème polynomiale déterministe.

ii. CLIQUE est un problème NP-complet.

Exercice I.3 *Soit A une matrice et b un vecteur à coefficients dans $\mathbf{Z}$. Le problème de programmation linéaire entière (ILP pour* integer linear programming*) consiste à déterminer s'il existe un vecteur $\vec{x}$ à coefficients dans $\mathbf{N}^m$ tel que $A\vec{x} = \vec{b}$.*[24] *Ce problème est dans NP. On utilise des notions d'algèbre linéaire pour montrer que si le problème a une solution alors il en a une dont la taille est polynomiale dans la taille de la matrice A. Ensuite on peut appliquer la méthode standard qui consiste à deviner un vecteur $\vec{x}$ et à vérifier qu'il est une solution. A partir de ce fait, le but de l'exercice est de montrer que le problème est NP-complet par réduction du problème SAT. Il peut être utile de considérer d'abord les problèmes suivants.*

- *Montrez qu'en introduisant des variables auxiliaires on peut exprimer la satisfaction d'une contrainte d'inégalité comme un problème d'ILP.*
- *Montrez qu'on peut exprimer la contrainte $x \in \{0, 1\}$.*
- *Montrez qu'on peut exprimer la contrainte $x = \overline{y}$ où $x, y \in \{0, 1\}$, $\overline{0} = 1$ et $\overline{1} = 0$.*
- *Montrez comment coder la validité d'une clause (disjonction de littéraux).*

[24] Comme pour le problème du voyageur de commerce, le problème ILP est souvent formulé comme un problème d'*optimisation*. Par exemple, il s'agit de minimiser une fonction linéaire $\vec{c}^T\vec{x}$ sous les contraintes $A\vec{x} = \vec{b}$ et $\vec{x} \geq 0$.

J TD : Preuves par induction

Exercice J.1 (transitivité) *Soit R une relation binaire sur un ensemble. Sa clôture réflexive et transitive R^* est la plus petite relation qui contient la relation identité, la relation R et telle que si $(x, y), (y, z) \in R^*$ alors $(x, z) \in R^*$. Montrez que R^* peut être vu comme un ensemble défini inductivement.*

Exercice J.2 *Un graphe non-dirigé G est composé d'un ensemble fini non-vide de noeuds N et d'un ensemble A d'arêtes qui connectent les noeuds. Formellement, une arête est un ensemble $\{i, j\}$ de noeuds de cardinalité 2. Le* degré *d'un noeud i dans un graphe est le nombre d'arêtes qui le contiennent. Par exemple, un noeud isolé a degré 0. Démontrez en utilisant le principe de récurrence l'assertion suivante :*

Chaque graphe avec au moins 2 noeuds contient 2 noeuds avec le même degré.

Exercice J.3 *Soit $(\mathbf{N} \cup \{\infty\}, \leq)$ l'ensemble des nombres naturels avec un élément maximum ∞, $0 < 1 < 2 < \ldots < \infty$. Montrez que toute fonction monotone f sur cet ordre admet un point fixe, c'est-à-dire un élément x tel que $f(x) = x$.*

Exercice J.4 *On considère l'ensemble de symboles fonctionnels*

$$\Sigma = \{\epsilon, a, b\}$$

où $ar(\epsilon) = 0$ et $ar(a) = ar(b) = 1$. Calculez l'ensemble librement engendré associé à Σ. Cet ensemble est-il isomorphe à un ensemble déjà considéré dans le cours ?

Exercice J.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 $k \leq min(n, m)$ $(x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, x_k < y_k)$ Est-il vrai que $<$ est un ordre bien fondé ? Donner soit une preuve soit un contre-exemple.*

K TD : Terminaison 1

Exercice K.1 *On considère des programmes impératifs* while *dont les variables prennent comme valeurs des nombres naturels. Montrez que le programme suivant termine où l'on sait que le test Φ termine et n'a pas d'effet de bord (c'est-à-dire que l'évaluation du test n'affecte pas la valeur associée aux variables) :*

$$\begin{array}{l} \textit{while } u > l+1 \textit{ do} \\ (r := (u+l) \textit{ div } 2; \\ \textit{if } \Phi \textit{ then } u := r \textit{ else } l := r) \end{array}$$

Exercice K.2 *Soit $(\{a,b,c,d\}, \rightarrow)$ un système de réécriture où $\rightarrow = \{(c,a),(c,d),(d,c),(d,b)\}$. Dire si le système termine, est localement confluent, est confluent.*

Exercice K.3 (1) *Utilisez le principe d'induction pour démontrer la terminaison de la fonction récursive a telle que :*

$$\begin{array}{ll} a(0,n) & = n+1 \\ a(m+1,0) & = a(m,1) \\ a(m+1,n+1) & = a(m,a(m+1,n)) \end{array}$$

(2) *Calculez à l'aide d'un programme autant de valeurs $a(n,n)$ que possible.*

Exercice K.4 *On étend l'ordre lexicographique à un produit $A = A_1 \times \cdots \times A_n$, $n \geq 3$, d'ordres bien fondés $(A_i, >_i)$:*

$$(x_1,\ldots,x_n) > (y_1,\ldots,y_n) \textit{ si } \exists\, k \leq n \ \ (x_1 = y_1, \ldots, x_{k-1} = y_{k-1} \textit{ et } x_k >_k y_k)$$

Montrez que $(A, >)$ est bien fondé.

Exercice K.5 (ordre produit) *Soient $(A_i, >_i)$ pour $i = 1,\ldots,n$ des ordres bien fondés. On définit une relation $>$ sur le produit cartésien $A_1 \times \cdots \times A_n$ par*

$$(a_1,\ldots,a_n) > (a'_1,\ldots,a'_n) \textit{ si } a_i \geq_i a'_i, i = 1,\ldots,n \textit{ et } \exists\, i \in \{1,\ldots,n\} \ \ a_i >_i a'_i$$

(1) *La relation $>$ est-elle un ordre bien fondé ?*

(2) *Comparez la relation $>$ à l'ordre lexicographique sur le produit défini dans l'exercice K.4.*

L TD : Terminaison 2

Exercice L.1 *Considérons les programmes while :*

$$\begin{array}{l} \textit{while } m \neq n \textit{ do} \\ \quad \textit{if } m > n \textit{ then } m := m - n \textit{ else } n := n - m \end{array}$$

$$\begin{array}{l} \textit{while } m \neq n \textit{ do} \\ \quad \textit{if } m > n \textit{ then } m := m - n \\ \quad \textit{else } h := m; m := n; n := h \end{array}$$

Dire si les programmes terminent quand les variables varient sur les nombres naturels positifs.

Exercice L.2 *Soit $A = \{a, b\}^*$ l'ensemble des mots finis sur l'alphabet $\{a, b\}$. Soit $\rightarrow$ une relation binaire sur A telle que :*

$$w \rightarrow w' \textit{ ssi } w = w_1 abw_2 \textit{ et } w' = w_1 bbaw_2$$

Donc w se réduit à w' si w' est obtenu de w en remplaçant un sous-mot ab avec bba.

Montrez ou invalidez les assertions suivantes (il est conseillé de s'appuyer sur les résultats démontrés dans le cours) :

1. *Le système de réduction $(A, \rightarrow)$ est à branchement fini.*
2. *Le système termine.*
3. *Le système est localement confluent.*
4. *Le système est confluent.*

Exercice* L.3 *Soient $X, Y \in \mathcal{M}(A)$. On écrit $X >_1 Y$ si Y est obtenu de X en remplaçant un élément de X par un multi-ensemble d'éléments strictement plus petits. Montrez que la clôture transitive de $>_1$ est égale à $>_{\mathcal{M}(A)}$.*

Exercice L.4 *Soit $\mathbf{N}^*$ les mots finis de nombres naturels. Montrez la terminaison du système :*

$$u(i+1)v \rightarrow iviui \textit{ pour } u, v \in \mathbf{N}^*$$

M TP : Méthode de Davis-Putnam

M.1 Objectifs

Le but de ce TP est d'implanter en langage Java la procédure de Davis-Putnam : vous devez réaliser un programme qui prend en entrée une formule A en forme normale conjonctive, puis décide si cette formule est satisfiable et si c'est le cas, renvoie une interprétation qui satisfait A. Le programme que vous allez réaliser va lire les formules à traiter dans un fichier. Ce fichier respecte un format particulier : le format DIMACS. Le choix des structures de données à employer est de votre ressort. Il est fortement conseillé de bien réfléchir à l'intégralité de l'algorithme avant d'implémenter les classes et les méthodes dont vous aurez besoin.

M.2 Définitions

Rappelons quelques définitions :
- un littéral est une variable propositionnelle x ou sa négation $\neg x$;
- une clause est une disjonction de littéraux ;
- une clause est une *tautologie* si et seulement si elle contient une variable x et sa négation $\neg x$.
- une clause est dite *unitaire* si elle contient exactement un littéral,
- une formule en forme normale conjonctive est une conjonction de clauses,
- une affectation v est une fonction partielle des variables aux valeurs booléennes.

M.3 Format DIMACS (http ://www.satlib.org/Benchmarks/SAT/satformat.ps)

Par exemple, la formule :

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

peut être codée par :

```
c Exemple fichier au format CNF
p cnf 4 3
1 3 -4 0
4 0
2 -3
```

- la ligne c est une ligne commentaire,
- la ligne p spécifie qu'il s'agit d'une formule en CNF avec 4 variables et 3 clauses,
- les lignes suivantes spécifient les clauses. Le littéral x_i est codé par i et le littéral $\neg x_i$ par $-i$ où $i \geq 1$ (et dans ce cas $i \leq 4$),
- les clauses peuvent être sur plusieurs lignes et elles sont séparées par 0.

Dans la page du cours, nous fournissons les fonctions **afficheDimacs** et **ecrisDimacs**, qui sont un exemple de lecture et d'écriture de fichiers DIMACS. Vous pourrez vous baser sur ces exemples pour réaliser l'interface de votre programme.

La réalisation de l'algorithme de Davis-Putnam exige la manipulation de formules, de clauses et d'affectations. Vous devrez donc définir les classes correspondantes ainsi que les méthodes dont vous aurez besoin. Voici quelques méthodes de base (il s'agit de simples suggestions, certaines méthodes pourront être omises ou ajoutées selon vos besoins).

Classe formule :
- un constructeur qui lit une formule CNF en format DIMACS et construit la formule correspondante,
- **vide** qui teste si la formule est vide,
- **affiche** qui écrit une formule dans un fichier au format DIMACS,
- **verifie** qui prend en argument une affectation et qui renvoie **true** si la formule est vraie dans cette affectation.

Classe clause :
- **appartient** qui prend en argument un littéral et qui renvoie **true** s'il apparaît dans la clause,
- **unitaire** qui renvoie **true** si la clause ne contient qu'un seul littéral,
- **vide** qui renvoie **true** si la clause est vide,
- **verifie** qui prend en argument une affectation, et qui renvoie **true** si la clause est vraie dans cette affectation.

Classe affectation :
- **fixe** qui prend en argument un littéral et un booléen et qui ajoute le littéral à l'affectation avec la valeur du booléen,
- **valeur** qui prend en argument un littéral et renvoie sa valeur dans l'affectation.

M.4 Davis-Putnam

La méthode de Davis-Putnam permet de décider si une formule en forme normale conjonctive est satisfiable. On représente une formule A en CNF comme un *ensemble* (éventuellement vide) de clauses $\{C_1, \ldots, C_n\}$ et une clause C comme un *ensemble* (éventuellement vide) de littéraux. Dans cette représentation, on définit la substitution $[b/x]A$ d'une valeur booléenne b dans A comme suit :

$$[b/x]A = \{[b/x]C \mid C \in A \text{ et } [b/x]C \neq 1\}$$

$$[b/x]C = \begin{cases} 1 & \text{si } (b = 1 \text{ et } x \in C) \text{ ou } (b = 0 \text{ et } \neg x \in C) \\ C \backslash \{\ell\} & \text{si } (b = 1 \text{ et } \ell = \neg x \in C) \text{ ou } (b = 0 \text{ et } \ell = x \in C) \\ C & \text{autrement} \end{cases}$$

La méthode de Davis-Putnam fonctionne comme suit. Au départ, A est une formule CNF :

- si A est vide, retourner true.
- si A contient la clause vide, retourner false.
- si A contient une clause C qui contient à la fois les littéraux x et $\neg x$, appeler la fonction davis-putnam sur la formule $A \setminus C$.
- si A contient une clause $\{x\}$ (resp. $\{\neg x\}$), appeler la fonction davis-putnam sur $[1/x]A$ (resp. $[0/x]A$).
- sinon, choisir une variable x dans A. Appliquer la procédure DP récursivement sur $[1/x]A$ et $[0/x]A$. Retourner true si l'un des résultats est true, retourner false sinon.

Vous devrez définir des méthodes pour chacune de ces opérations. La dernière, en particulier, doit être traitée avec attention : si la première affectation choisie échoue, il faut pouvoir revenir à l'état courant pour tester la deuxième ; une forme de sauvegarde ou de duplication sera donc nécessaire.

Exercice M.1 *1- Programmez une méthode estSatisfiable qui décide si la formule est satisfiable en utilisant la procédure de Davis-Putnam*
2- Modifiez la fonction estSatisfiable pour que si la formule est satisfiable, elle affiche une affectation v qui satisfait A.

M.5 Test

Il s'agit maintenant de tester la correction et l'efficacité de votre programme.

- Il est facile de vérifier si une formule A est satisfiable par une affectation v.

  Exercice M.2 *Programmez une méthode permettant ce test.*

- Il est plus compliqué de vérifier qu'une formule n'est pas satisfiable. Une possibilité est de générer de façon aléatoire un certain nombre d'affectations et de vérifier qu'elles ne satisfont pas la formule.

  Exercice M.3 *Programmez une méthode qui réalise ce test sur une centaine d'affectations prises au hasard.*

- Sur quelles formules tester votre programme ? Il est pratique de disposer d'un générateur de formules. Par exemple, on peut programmer une fonction $G(n, m, p)$ qui génère une formule avec n clauses et m variables avec la propriété que :
  - le littéral x_j est présent dans la clause C_i avec probabilité $p/2$;
  - le littéral $\neg x_j$ est présent dans la clause C_i avec probabilité $p/2$;
  - les littéraux x_j et $\neg x_j$ ne sont jamais présents ensemble dans une clause C_i (et donc ils sont absents avec probabilité $(1 - p)$).

  Exercice M.4 *Programmez une procédure qui prend les paramètres (n, m, p, k), génère k formules en utilisant la fonction $G(n, m, p)$, applique la procédure DP pour déterminer la satisfiabilité et applique les méthodes développées dans les exercices M.2 et M.3 pour vérifier le résultat.*

M.6 Heuristique

Le choix d'une variable x dans la dernière étape peut avoir beaucoup d'influence sur la rapidité de la procédure. Une heuristique possible est de choisir x de sorte que le nombre de clauses dans lesquelles x apparaît multiplié par le nombre de clauses dans lesquelles $\neg x$ apparaît est maximal, et de tester $DP([1/x]A)$ d'abord s'il y a plus de clauses contenant x que de clauses contenant $\neg x$, et $DP([0/x]A)$ sinon.

Exercice M.5

1. *Implanter cette stratégie dans la fonction DP. Soit DPH la fonction obtenue.*
2. *Modifiez le code de la fonction DP et de la fonction DPH pour qu'elles retournent le nombre de fois que le pas de 'choix' $DP([0/x]A)$ or $DP([1/x]A)$ est exécuté.*
3. *Programmez une procédure qui prend les paramètres (n, m, p, k), génère k formules en utilisant la fonction $G(n, m, p)$, applique les procédures DP et DPH aux formules, vérifie*

qu'elles produisent le même résultat (satisfiable ou pas satisifiable) et pour chaque formule imprime le nombre de fois que le pas de 'choix' est exécuté par DP et DPH.

M.7 Le principe du pigeonnier (*pigeon principle*)

On dispose de m pigeons et de n nids. Le problème $P(m, n)$ a une solution si :
- Chaque pigeon a un nid.
- Chaque nid contient au plus un pigeon.

Il est évident que le problème n'a de solution que si m est inférieur ou égal à n, mais la vérification de ce fait par la méthode de Davis-Putnam peut s'avérer très coûteuse.

On écrit le problème du pigeonnier en CNF de la façon suivante :
- On introduit les variables $o_{i,j}$ pour $i \in [1..m], j \in [1..n]$. On interprète $o_{i,j}$ par *le pigeon i occupe le nid j.*
- On introduit la CNF
$$\bigwedge_{i=1,\ldots,m} \Big(\bigvee_{j=1,\ldots,n} o_{i,j} \Big)$$
qui exprime le fait que chaque pigeon doit se trouver dans un nid.
- On introduit la CNF
$$\bigwedge_{j=1,\ldots,n,\ i,k=1,\ldots,m,i<k} (\neg o_{i,j} \vee \neg o_{k,j})$$
qui exprime le fait que dans chaque nid il y a au plus un pigeon.
- On prend la conjonction des CNF (qui est encore une CNF !).

Exercice M.6 *1. Programmez une fonction* **pigeon** *qui prend en argument deux entiers (m, n), et qui produit un fichier DIMACS contenant la formule CNF qui correspond au problème $P(m, n)$.*

2. Testez vos programmes DP et DPH pour voir jusqu'à quelle valeur de m ils parviennent à résoudre les problèmes $P(m, m)$ et $P(m + 1, m)$.

N TP : Sudoku

N.1 Objectifs

Le but de ce TP est de résoudre un problème combinatoire, le Sudoku, en le traduisant vers un problème de satisfiabilité. On utilisera ensuite un solveur pour résoudre ce problème de satisfiabilité.

N.2 Le solveur SATO

Le programme SATO est un solveur qui prend comme entrée une formule CNF dans un fichier sous forme DIMACS et retourne, si elle existe, une affectation qui satisfait cette formule.

Avant de commencer, vous devez installer ce programme sur votre compte. Pour cela, vous devez :

- récupérer le fichier `sato4.2.tgz` à l'adresse suivante :
  `http ://www.cs.uiowa.edu/~hzang/sato.html`
- décompresser l'archive en exécutant la commande :
  `> tar xvf sato4.2.tgz`
- compiler le programme en exécutant dans le dossier de SATO la commande :
  `> make`

Plus d'informations sont disponibles dans le fichier `README` à la même adresse.

Vous pouvez tester le solveur en utilisant par exemple le générateur programmé dans le TP1 et comparer les performances avec votre implémentation de l'algorithme de Davis-Putnam.

N.3 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.

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.

L'entrée de votre programme est un fichier qui représente la grille sous la forme d'une matrice 9×9. Lorsque le chiffre d'une case n'est pas défini, on note 0. Ainsi l'exemple précédent est représenté dans ce fichier sous la forme :

```
5 3 0 0 7 0 9 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 0 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

N.4 Interface

La principale tâche à effectuer est l'interfaçage entre le Sudoku et SATO. Dans une premier temps on doit traduire le problème du Sudoku vers un problème de satisfiabilité. Puis on doit transformer une affectation qui satisfait la formule générée en une solution du Sudoku.

Notez qu'en Java, on ne peut faire d'appel système, il faudra donc exécuter le programme final à l'aide d'un script de la forme :

```
java Sudoku2SAT fic_entree
./sato fic_dimacs > fic_sol
java SAT2Sudoku fic_sol
```

Où `fic_entree` est le fichier qui représente la grille d'entrée, `fic_dimacs` est la traduction du problème en format DIMACS produit par le programme `Sudoku2SAT` et `SAT2Sudoku` traduit l'affectation retournée par SATO en solution du problème initial. Le but du TP est de programmer `Sudoku2SAT` et `SAT2Sudoku`.

N.5 Sudoku → SATO

Pour chaque case (x, y) de la grille (ligne x et colonne y) et chaque valeur $z \in [1..9]$ on introduit une variable propositionnelle s_{xyz}. Le problème du Sudoku peut être traduit vers une formule CNF de la façon suivante :

- Tout d'abord on veut que chaque case (x, y) contienne au moins un chiffre entre 1 9. Par exemple, pour la case $(1, 1)$ la clause générée sera :

$$s_{111} \vee s_{112} \vee s_{113} \vee \ldots \vee s_{119}.$$

- 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 à l'ensemble de clauses :

$$(\neg s_{111} \vee \neg s_{121}) \wedge (\neg s_{111} \vee \neg s_{131}) \wedge \ldots \wedge (\neg s_{111} \vee \neg s_{191}) \wedge (\neg s_{121} \vee \neg s_{131}) \wedge \ldots$$

- 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 avec un seul littéral.

Notez que si on s'arrête là le codage est suffisant, mais on peut ajouter les contraintes suivantes :

- Chaque case (x, y) contient au plus un chiffre entre 1 et 9.
- Chaque chiffre apparaît au moins une fois dans chaque ligne.
- Chaque chiffre apparaît au moins une fois dans chaque colonne.
- Chaque chiffre apparaît au moins une fois dans chaque région.

Exercice N.1 *Écrire un programme qui traduit un problème de Sudoku en une formule CNF sous le format DIMACS. Ce programme prend comme entrée un fichier sous la forme définie dans la partie précédente et doit écrire le résultat de la traduction dans un fichier de sortie.*

N.6 SATO → Sudoku

Il reste à transformer une affectation renvoyée par le solveur en solution pour le Sudoku.

Exercice N.2 *Écrire, une fonction qui transforme une affectation qui satisfait la formule en solution du Sudoku et l'affiche.*

Vous pouvez maintenant tester votre programme en essayant de résoudre des grilles proposées par exemple aux adresses suivantes :

```
http ://logiciel.sudoku.free.fr/
http ://www.esudoku.fr/
...
```

Vous pouvez aussi tester dans quelle mesure l'ajout de contraintes supplémentaires améliore les performances du programme.

O TP : Résolution

Pour représenter les formules en CNF on adopte la même notation ensembliste utilisée pour décrire la méthode de Davis-Putnam.

– Une clause C est un ensemble de littéraux.
– Une formule A en CNF est un ensemble de clauses.

Soient $C \cup \{x\}$ et $C' \cup \{\neg x\}$ deux clauses où l'on suppose que $x \notin C$ et $\neg x \notin C'$.

On dit que $C \cup C'$ est un *résolvant* des deux clauses.

La méthode de résolution peut être formulée de la façon suivante. Soit A un ensemble *fini* de clauses. Si X est un ensemble de clauses, on pose

$$\mathcal{F}_A(X) = A \cup X \cup \{C \mid C \text{ est un résolvant de deux clauses dans } X\}$$

Soit $Res(A)$ le plus petit point fixe de $\mathcal{F}_A$. On peut montrer que la formule A est insatisfiable si et seulement si $Res(A)$ contient la clause vide.

Exercice O.1 *Montrez (sur papier) que $Res(A)$ est fini et peut être calculé.*

On rappelle qu'une *clause de Horn* est une clause qui contient au plus un littéral positif. Par ailleurs, une *clause unitaire* est une clause qui contient un littéral. Dans la *méthode de résolution unitaire* on se limite à calculer les résolvants de couples de clauses dont au moins une est unitaire. On peut montrer qu'une conjonction de clauses de Horn n'est pas satisfiable si et seulement si la méthode de résolution *unitaire* dérive la clause vide.

Exercice O.2 *Construire un programme qui reçoit en entrée une formule A qui est une conjonction de clauses de Horn au format dimacs et vérifie si A est insatisfiable en utilisant la méthode de résolution unitaire. Le programme imprime $Res(A)$ (adapté pour la résolution unitaire) s'il n'arrive pas à générer la clause vide.*

Exercice O.3 *Estimez (sur papier) la complexité de votre programme en fonction de la taille de la formule A.*