N° d'ordre : 08 / 2006 - M / IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L4ENSEIGNEMENT SUPEREUR ET DE LA RECHERCHE SCIENTIFQUE

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Electronique et d'Informatique

MEMOIRE

Présenté pour l'obtention du diplôme de MAGISTER

En : Informatique

Spécialité I.A.B.D.A.

Par : Melle **YAHI Safa**

Sujet

Compilation de bases de croyances possibilistes

Soutenu le 18 / 07 / 2006, devant le jury composé de :

Mme	**A. AISSANI**	Professeur	U.S.T.H.B	Présidente
Mme	**H. DRIAS**	Professeur	U.S.T.H.B / I.N.I	Directrice de thèse
Mr	**S. BENFERHAT**	Professeur	Université d'Artois	Examinateur
Mr	**H. AZZOUNE**	Maître de conférence	U.S.T.H.B	Examinateur
Mme	**F. KHELLAF**	Chargé de cours	U.S.T.H.B	Examinatrice

Introduction générale

La représentation des connaissances et le raisonnement à partir de celles-ci constituent l'épine dorsale de l'Intelligence Artificielle. En général, pour représenter des connaissances, on fait recours à des formalismes logiques. En conjecturant que les connaissances en question sont certaines et cohérentes, la logique classique, en particulier la logique propositionnelle, s'avère satisfaisante. Néanmoins, d'un point de vue pratique, on se heurte à des problèmes de type complexité computationnelle et l'explosion combinatoire est inéluctable au pire des cas. En effet, tester la satisfiabilité d'une formule propositionnelle (problème SAT en abrégé) constitue le problème NP-complet archétype. D'autre part, vérifier si une formule est déductible à partir d'une autre ou non revient à un problème CoNP-complet.

Dans l'optique d'appréhender cet écueil, diverses solutions ont été proposées entre autres la compilation de connaissances. Il s'agit d'une technique relativement récente et qui s'est avérée assez efficace pour traiter de nombreuses instances de problèmes en pratique. Le principe sous-jacent est d'effectuer un prétraitement sur la base de connaissances en vue de générer une base compilée qui, à partir de laquelle, le raisonnement devient polynomial (ou plutôt plus efficace dans le cas général).

Initialement, la plupart des méthodes de compilation de connaissances en logique propositionnelle gravitaient autour d'un seul langage cible de compilation, largement étudié en informatique, à savoir le langage des impliqués premiers PI. De plus, elles ne prenaient en compte que la déduction logique. Récemment dans [Darwiche and Marquis 2002], Darwiche et Marquis proposent un répertoire considérable de langages cibles de compilation dérivés du langage NNF et qui sont issus de travaux en Intelligence Artificielle, en vérification formelle et en informatique de manière générale. Outre que le problème de déduction, ils considèrent un large spectre de requêtes et de transformations logiques fréquemment utilisées en Intelligence Artificielle. Ils s'intéressent également à un critère délicat qu'est la concision. Ainsi, étant donnée une application particulière, ils suggèrent d'opter pour le langage cible de compilation le plus concis qui permet d'effectuer en temps polynomial les opérations logiques auxquelles fait appel l'application en question. Dans ce contexte, le langage DNNF (pour Decomposable Negation Normal Form) suscite un intérêt particulier. Il permet d'effectuer en temps polynomial des opérations logiques suffisantes pour de nombreuses applications complexes. D'autre part, DNNF est plus concis que tous les autres langages cibles de compilation excepté PI. En fait, on sait que PI n'est pas plus concis que DNNF mais on ignore jusqu'à présent l'inverse.

En réalité, les connaissances dont nous disposons ne sont pas toujours certaines et cohérentes entre elles. Dans ce cas, la logique propositionnelle ou la logique classique en général est très vite inexploitable. D'autres formalismes ont été mis au point afin d'exprimer des connaissances incertaines ou contradictoires entre autres la logique possibiliste. Cette logique est une extension de la logique classique où syntaxiquement les formules (ou croyances) se voient assigner un degré de certitude. Sémantiquement, les contres modèles des formules les moins certaines sont préférés aux contres modèles des formules les plus certaines. Cette logique est de plus en plus utilisée en pratique à titre

d'exemple en sécurité informatique, dans les systèmes multi-agents et même en informatique médicale.

Toutefois, le gain d'expressivité offert par la logique possibiliste s'accompagne d'un saut de complexité. En logiques possibiliste, on considère deux relations d'inférence. Alors que la première relation constitue un problème BH_2-complet, la seconde est dans Δ_2^P [$O(\log n)$].

Dernièrement, Benferhat et Prade [Benferhat and Prade 2006] ont proposé un premier compilateur de bases de croyances possibilistes. Ce dernier s'appuie principalement sur un codage propositionnel de la base en question et sur les opérations d'oubli de variables et de littéraux. La clef de son efficacité réside dans l'utilisation du langage DNF.

Cependant, le langage DNF présente une limitation qui se trouve dans le fait qu'il existe des formules logiques qui n'admettent que des formules DNF équivalentes de taille exponentielle ce qui remet en cause l'efficacité du compilateur possibiliste DNF proposé par Benferhat et Prade.

Notre objectif alors est de proposer, en se basant sur ce compilateur DNF, d'autres compilateurs possibilistes plus efficaces et ce en tirant profit des travaux effectués par Darwiche et Marquis en compilation de connaissances propositionnelles. Le langage DNNF étant plus concis que le langage DNF (dans le sens où il existe des formules logiques n'admettant que des formules DNF équivalentes de taille exponentielle mais ayant des formules DNNF équivalentes de taille polynomiale), nous stimule à introduire un compilateur possibiliste DNNF. En revanche, il est probable que DNNF ne soit pas plus concis que PI d'où la pertinence d'introduire un compilateur PI en logique possibiliste. En outre, il nous semble intéressant de définir un compilateur possibiliste plus général en le paramétrant par un langage cible de compilation C satisfaisant certains critères précis. Hormis l'élégance d'un tel compilateur au niveau implémentation et même au niveau utilisation, ça permet d'exploiter d'une manière raffinée et aisée d'autres langages cibles de compilation outre que DNF, DNNF et PI en vue de définir de nouveau compilateurs possibilistes.

Plan du mémoire

Outre que l'introduction générale, le présent mémoire se veut structuré en quatre chapitres comme suit :

Dans le premier chapitre, nous esquissons les notions de base de la logiques propositionnelle en décrivant ses aspects syntaxiques et sémantiques essentiels. Nous esquissons en outre la théorie de la complexité en définissant la machine de Turing et certaines classes de complexité. Nous définissons également la hiérarchie polynomiale HP.

Dans le second chapitre, un large passage est dédié à la compilation de connaissances en logique propositionnelle. Nous commençons par donner une définition formelle de la compilation ainsi qu'aux problèmes compilables. Nous décrivons par la suite les méthodes classiques de compilation exacte et approximative. Un accent est mis sur la définition du langage NNF et ses différents dérivés, sur leur concision ainsi sur les opérations logiques qu'ils permettent d'effectuer en temps polynomial. Après cela, nous nous focalisons sur le langage DNNF étant données ses caractéristiques très intéressantes en

décrivant le compilateur propositionnel DNNF proposé par Darwiche [Darwiche 2001a]. Avant de conclure ce chapitre en évoquant d'autres formes de compilation de connaissances, nous exhibons le lien étroit existant entre le problème SAT et la compilation de connaissances.

Le troisième chapitre a pour objet la présentation de la logique possibiliste, sa syntaxe, sa sémantique, ses applications, ses relations d'inférence et leur complexité computationnelle. De plus, nous abordons le compilateur possibiliste DNF proposé par Benferhat et Prade.

Le quatrième chapitre quant à lui, il décrit notre contribution en exploitant les derniers travaux en matière de compilation de bases de connaissances propositionnelles et en matière de compilation de bases de croyances possibilistes. En fait, nous proposons un compilateur DNNF en logique possibiliste. Nous introduisons aussi un compilateur PI. De plus, nous définissons un compilateur plus général qui se veut paramétré par un langage cible de compilation C satisfaisant certaines conditions.

Enfin, nous terminons par une conclusion générale et quelques perspectives.

Chapitre 1

Logique propositionnelle & Théorie de la complexité

Sommaire

«La logique n'est pas une théorie, mais une image réfléchie du monde.»
Ludwig Wittgenstein

1.1 Logique propositionnelle

La logique propositionnelle, introduite sous sa forme moderne par G. Boole [Boole 1854] il y a presque 150 ans, constitue un formalisme de modélisation quelque peu rudimentaire. Cependant, de nombreuses questions liées à cet outil de modélisation de la connaissance et du raisonnement restent ouvertes et de multiples problèmes pratiques en informatique peuvent se représenter de manière simple et se résoudre en logique propositionnelle. En outre, elle constitue l'ossature des logiques modernes. Notre but ici n'est pas de faire un état de l'art complet sur la logique propositionnelle mais plutôt de présenter succinctement ses aspects syntaxiques et sémantiques essentiels pour la compréhension de ce mémoire.

1.1.1. Aspects syntaxiques

Définition 1.1 (Proposition)
Une **proposition** (appelée également **atome** ou **variable propositionnelle**) est une variable booléenne susceptible de prendre deux valeurs de vérité : Vrai (V) ou Faux (F).

Définition 1.2 (Connecteur logique)
Les connecteurs logiques servent à relier les propositions entre elles. On distingue :

- la négation : $\neg$;
- la conjonction : $\wedge$;
- la disjonction : $\vee$;
- l'implication matérielle : $\Rightarrow$;
- l'équivalence : $\Leftrightarrow$.

Les connecteurs $\wedge$, $\vee$, $\Rightarrow$ et $\Leftrightarrow$ sont de même priorité, l'ordre d'évaluation étant ainsi déterminé par les parenthèses. Le connecteur de négation est considéré quant à lui comme prioritaire sur tous les autres connecteurs.

Soit V un ensemble fini de variables propositionnelles. $PROP_V$ désigne le langage propositionnel construit à partir des variables de V, des constantes booléennes $\bot$ et $\top$ et des connecteurs logiques. Formellement :

Définition 1.3 ($\boldsymbol{PROP_V}$)
Soit V un ensemble fini de variables propositionnelles. $\boldsymbol{PROP_V}$ est le langage propositionnel défini inductivement par :

- $\bot$ et $\top \in PROP_V$;
- si $x \in V$ alors $x \in PROP_V$;
- si $f \in PROP_V$ alors $(f), \neg f \in PROP_V$;
- Si f et $g \in PROP_V$ alors $f \wedge g, f \vee g, f \Rightarrow g$ et $f \Leftrightarrow g \in PROP_V$.

Les élément de $PROP_V$ sont appelés **formules** (ou **formules propositionnelles**).

La taille d'une formule f , dénotée par $| f |$, est le nombre de symboles (incluant les variables, les connecteurs et les constantes booléennes) utilisés pour son écriture.

Exemple 1.1
$(a \vee b) \wedge ((c \Rightarrow d) \wedge \neg a)$ est une formule.

Définition 1.4 (Littéral)
Un **littéral** l est soit une variable propositionnelle x (on parle de littéral positif), soit sa négation $\neg x$ (on parle alors de littéral négatif).

L_V (resp. L_V+, L_V-) dénote l'ensemble des littéraux (resp. positifs, négatifs) construits sur V.

Définition 1.5 (Littéral complémentaire)
Soit l un littéral, $\neg l$ représente le **littéral complémentaire** de l.

Définition 1.6 (Littéral pur)
Un littéral l est dit **pur** (ou **monotone**) pour une formule f si l apparaît dans f et $\neg l$ n'apparaît pas.

Définition 1.7 (Clause)
Une **clause** c est une disjonction finie de littéraux (en particulier la constante $\bot$, lorsque l'ensemble des littéraux est vide).

Définition 1.8 (Terme)
Un **terme** t (**monôme** ou encore **cube**) est une conjonction finie de littéraux (en particulier la constante $\top$, lorsque l'ensemble des littéraux est vide).

Exemple 1.2
$a \vee b$ est une clause.
$\neg a \wedge b$ est un terme.

Définition 1.9 (Clause unitaire)
Une **clause unitaire** ou **unaire** est une clause qui contient un seul littéral.

Définition 1.10 (Clause de Krom)
Une **clause de Krom** est une clause qui contient au plus deux littéraux.

Définition 1.11 (Clause positive, négative, mixte)
On appelle **clause positive** (resp. **négative**) une clause qui ne contient que des littéraux positifs (resp. négatifs). Une clause **mixte** est une clause qui contient à la fois des littéraux positifs et négatifs.

Définition 1.12 (Clause de Horn)
Une **clause de Horn** est une clause qui contient au plus un littéral positif.

Définition 1.13 (Clause reverse-Horn)
Une **clause reverse-Horn** est une clause qui contient au plus un littéral négatif.

Définition 1.14 (Clause fondamentale, terme fondamental)
Une **clause fondamentale** (resp. terme fondamental) est une clause (resp. terme) qui ne contient pas de littéraux complémentaires.

Définition 1.15 (Clauses résolvantes)
Deux clauses c_1 et c_2 **se résolvent** si et seulement s'il existe un littéral l tel que $l \in c_1$ et $\neg l \in c_2$. La clause $((c_1 - \{l\}) \cup (c_2 - \{\neg l\}))$ est appelée **résolvante**, plus précisément **résolvante** en l de c_1 et c_2.

Exemple 1.3
a est une clause unitaire.
$a \vee b$ est une clause de Krom.
$a \vee \neg b \vee \neg c$ est une clause de Horn.
$a \vee b \vee \neg c$ est une clause reverse-Horn.
$a \vee b \vee \neg c$ est une clause fondamentale.
Les clauses $(a \vee b)$ et $(\neg b \vee c)$ se résolvent en la clause $a \vee c$.

Définition 1.16 (Var, Lit)
Soit f une formule propositionnelle. Var(f) (resp. Lit(f)) désigne l'ensemble des variables (resp. littéraux) qui apparaissent dans f.

Définition 1.17 (Sous-sommation)
Une clause c_1 **sous-somme** ou **subsume** une clause c_2 si et seulement si $\text{Lit}(c_1) \subseteq \text{Lit}(c_2)$.

Exemple 1.4
$a \vee b$ sous somme $a \vee b \vee \neg c$.

Définition 1.18 (CNF)
Une formule f est sous **forme normale conjonctive (CNF)** si et seulement si f est une conjonction de clauses.

Définition 1.19 (DNF)
Une formule f est sous **forme normale disjonctive (DNF)** si et seulement si f est une disjonction de termes.

Définition 1.20 (Formule de Horn CNF, Formule reverse-Horn CNF)
Une **formule de Horn CNF** (resp. **reverse-Horn CNF**) est une formule CNF composée uniquement de clauses de Horn (resp. reverse-Horn).

Définition 1.21 (Base)
Une **base** est un ensemble fini de formules que l'on assimile à la conjonction de ses formules.

Exemple 1.5
$(a \vee b) \wedge (\neg c \vee d)$ est une formule CNF.
$(a \wedge b) \vee (\neg c \vee d)$ est une formule DNF.
$(\neg a \vee \neg b) \wedge (\neg c \vee d)$ est une formule de Horn CNF.
$(a \vee b) \wedge (\neg c \vee d)$ est une formule reverse-Horn CNF.

1.1.2. Aspects sémantiques

Définition 1.22 (Interprétation)
Une **interprétation** I est une application de l'ensemble des formules propositionnelles dans l'ensemble des valeurs de vérité {V, F}.
L'interprétation $I(f)$ d'une formule f se calcule via les règles suivantes :

- $I(\top) = \text{V}$;
- $I(\bot) = \text{F}$;
- $I(\neg f) = \text{V}$ si et seulement si $I(f) = \text{F}$;
- $I(f \wedge g) = \text{V}$ si et seulement si $I(f) = I(g) = \text{V}$;
- $I(f \vee g) = \text{F}$ si et seulement si $I(f) = I(g) = \text{F}$;
- $I(f \Rightarrow g) = \text{F}$ si et seulement si $I(f) = \text{V}$ et $I(g) = \text{F}$;
- $I(f \Leftrightarrow g) = \text{V}$ si et seulement si $I(f) = I(g)$.

La table suivante synthétise les valeurs de vérités de certaines formules en fonction des valeurs de vérité de deux formules f et g :

$I(f)$	$I(g)$	$I(\neg f)$	$I(f \wedge g)$	$I(f \vee g)$	$I(f \Rightarrow g)$	$I(f \Leftrightarrow g)$
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	V	V

Définition 1.23 (Modèle)
On dit qu'une interprétation I est un **modèle** d'une formule f si $I(f) = V$. On dit aussi que I **satisfait** f.

Définition 1.24 (Contre-modèle)
On dit qu'une interprétation I est un **contre-modèle** d'une formule f si $I(f) = F$. On dit aussi que I **falsifie** (**contredit**) f.

Définition 1.25 (Consistance)
Une formule est dite **consistante** (**satisfiable** ou **cohérente**) si et seulement si elle admet au moins un modèle.

Définition 1.26 (Inconsistance)
Une formule est dite **inconsistante** (**insatisfiable** ou **contradictoire**) si et seulement si elle n'admet pas de modèle.

Définition 1.27 (Validité)
Une formule est dite **valide** (**universellement valide** ou **tautologie**) si et seulement si elle n'admet pas de contre-modèle.

Définition 1.28 (Invalidité)
Une formule est dite **invalide** si et seulement si elle admet un contre-modèle.

Exemple 1.6
$a \vee \neg a \vee b$ est valide.
$a \wedge b$ est invalide mais elle est consistante.
$a \wedge b \wedge (\neg a \vee \neg b)$ est inconsistante.

On définit maintenant la relation d'*inférence* ou de *déduction* en logique propositionnelle :

Définition 1.29 (Conséquence logique)
Une formule f **implique sémantiquement** une formule g, qu'on note par $f \models g$, si et seulement si tout modèle de f est un modèle de g. On dit aussi que g est une **conséquence logique** (**inférable** ou **déductible**) de f.

Exemple 1.7
$a \wedge b \models a$
$a \models a \vee b$

Il est à noter que si c_1 et c_2 sont deux clauses fondamentales, alors $\text{Lit}(c_1) \subseteq \text{Lit}(c_2)$ si et seulement si $c_1 \models c_2$.

Définition 1.30 (Equivalence)
On dit que deux formules f et g sont **logiquement équivalentes** et on note $f \equiv g$ si et seulement si $f \models g$ et $g \models f$, c'est-à-dire elles admettent exactement les mêmes modèles.

Exemple 1.8
$\neg f \vee g \equiv f \Rightarrow g$
$(f \Rightarrow g) \wedge (f \Rightarrow g) \equiv f \Leftrightarrow g$

Définition 1.31 (Impliqué)
Une clause c est un **impliqué** d'une formule f si et seulement si $f \models c$.

Définition 1.32 (Impliquant)
Un terme t est un **impliquant** d'une formule f si et seulement si $t \models f$.

Définition 1.33 (Impliqué premier)
Une clause c est un **impliqué premier** d'une formule f si et seulement si

- c est un impliqué de f et
- pour tout autre impliqué c' de f, si $c' \models c$, alors $c \models c'$.

Définition 1.34 (Impliquant premier)
Une terme t est un **impliquant premier** d'une formule f si et seulement si

- t est un impliquant de f et
- pour tout autre impliquant t' de f, si $t \models t'$, alors $t' \models t$.

On note que les impliqués premiers et les impliquants premiers sont minimaux pour l'inclusion.

Exemple 1.9
Soit $f = (a \vee \neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg a \vee \neg d) \wedge (b \vee c)$.
$\neg a \vee \neg c \vee d$ est impliqué de f ;
$\neg a \vee \neg c$ est un impliqué premier de f ;
$\neg a \wedge b \wedge c \wedge d$ est un impliquant de f ;
$\neg a \wedge c \wedge d$ est impliquant premier de f.

1.2. Théorie de la complexité

La notion de complexité revêt une importance capitale en informatique. La théorie de la complexité est une vaste théorie dont tous les aspects n'ont pas encore été explorés. Dans cette partie, on présente

les bases de la théorie de la complexité. La définition des différentes classes de complexité se basant sur un modèle de machine appelé *machine de Turing*, il convient de définir ce modèle de calcul dans un premier temps. Ensuite, on définit quelques classes de complexité. Enfin, on présente la hiérarchie polynomiale.

1.2.1. Machine de Turing

Une machine de Turing est un modèle de calcul crée par Alan Turing en vue de donner une définition précise au concept d'algorithme. La thèse de Church-Turing postule que tout procédé de calcul de type algorithmique correspond à une machine de Turing. Ce modèle bien que simple et primitif est capable d'exprimer toute procédure de calcul.

Définition 1.35 (Machin de TURING)
Une **machine de TURING** est définie par la donnée des éléments suivants :

1. Une *bande* de longueur infinie, découpée en cases où chaque case contient un symbole d'un alphabet fini. L'alphabet contient un symbole spécial 'blanc' et un ou plusieurs autres symboles.
2. une *tête de lecture/écriture* pouvant lire et écrire des symboles sur la bande et se déplacer d'une case à la fois, vers la gauche ou vers la droite ;
3. un *registre d'état* qui mémorise l'état courant de la machine. Le nombre d'états possibles est toujours fini. On distingue des états spéciaux : l'états initial et le(s) état (s) d'arrêt.
4. une *table des transitions* qui indique quel symbole écrire, comment déplacer la tête et quel est le nouvel état en fonction de l'état courant de la machine et du symbole lu sur la bande.

Un calcul sur une machine de TURING s'effectue de la manière suivante :

- la donnée d'entrée est placée sur la bande, la tête pointant sur son début ;
- le calcul se déroule suivant la table des transitions et se termine lorsqu'une transition conduit à un état d'arrêt ;
- le résultat peut alors être lu sur la bande.

Il est possible que la machine n'atteigne jamais un état d'arrêt, on dit alors qu'elle boucle.

Définition 1.36 (Machine de TURING déterministe, non déterministe)
Une machine de TURING est dite **déterministe** si l'ensemble des transitions qui lui est associé est constitué des transitions déterministes. A l'inverse, une machine de TURING **non déterministe** possède au moins une transition non déterministe dans sa table des transitions.

Les ordinateurs actuels sont conçus d'après un modèle de machine à accès aléatoire. Les machines à accès aléatoire possèdent une mémoire adressable d'où un accès direct à l'information, contrairement à la machine de TURING qui est obligée de parcourir séquentiellement sa bande pour obtenir un résultat intermédiaire. Il est donc naturel de se demander pourquoi baser nos définitions sur un modèle aussi rudimentaire que la machine de TURING. Cependant, il est possible de montrer que tout calcul nécessitant un temps en $f(n)$ sur une machine à accès aléatoire, peut être traduit sur une machine de TURING effectuent le même calcul en $O(f^6(n))$; voir même $O(f^3(n))$ avec une machine de TURING à plusieurs rubans [Papadimitriou 1994].

Malgré cette augmentation de temps, non négligeable, cette propriété a permis d'utiliser la machine de TURING comme référence dans les définitions théoriques des classes de complexité.

1.2.2. Classes de complexité

On note d'abord que les problèmes auxquels on s'intéresse sont des problèmes de *décision* :

Définition 1.37 (Problème de décision)
Un problème de **décision** *P* est un problème n'ayant que deux solutions possibles: 'OUI' ou 'NON'. En d'autre terme, l'ensemble D_P des instances de *P* est scindé en deux ensembles disjoints :

- Y_p représentant l'ensemble des instances positives c'est-à-dire les instances pour lesquelles la réponse est par 'OUI'.
- N_p représentant l'ensemble des instances négatives c'est-à-dire les instances pour lesquelles la réponse est par 'NON'.

On définit le problème *complémentaire* d'un problème de décision comme suit :

Définition 1.38 (problème complémentaire)
Le problème **complémentaire** d'un problème de décision *P* est le problème de décision noté Co*P* tel que : $D_{CoP} = Dp$ et $Y_{CoP} = N_P$.

Afin d'analyser et de comparer en terme de performance les algorithmes dédiés à la résolution d'un problème donné, deux critères naturels sont à considérer : le temps et l'espace mémoire d'où la notion de complexité.

Définition 1.39 (Complexité)
La **complexité temporelle** d'un algorithme est le nombre d'opérations élémentaires qu'il effectue. Sa **complexité spatiale** est la taille de l'espace mémoire qu'il requiert pour son déroulement.

En particulier, dans le cadre de la machine de TURING, la complexité temporelle correspond au nombre de transitions à effectuer. Quant à la complexité spatiale, elle correspond au nombre de cases de la bande à utiliser.

La complexité est fonction de la taille de la donnée d'entrée. Dire qu'un algorithme est en $O(f(n))$ signifie que sa complexité temporelle est bornée par $c*f(n)$ lorsque n tend vers l'infini où c est une constante réelle et n est la taille de la donnée d'entrée. Asymptotiquement, un algorithme en $O(n \log n)$ est plus performant qu'un algorithme en $O(n^2)$, lequel est beaucoup plus performant qu'un algorithme en $O(2^n)$.

Exemple 1.10
Soit l'algorithme qui fait la multiplication de deux matrices carrées d'ordre *n*, on prend comme opérations élémentaires : l'addition et la multiplication. La complexité temporelle C_t de cet algorithme est donnée par :
$C_t = n^2 (n (multi) + (n-1) (add)) \rightarrow n^3$ *lorsque* $n \rightarrow$ *infini* .
On dit alors que cet algorithme est en $O(n^3)$.

Les algorithmes sont classés, selon leur complexité, en deux classes : les algorithmes polynomiaux (efficaces) et les algorithmes exponentiels.

Définition 1.40 (Algorithme polynomial, exponentiel)
Un algorithme est dit **polynomial** s'il est en $O(P(n))$, P étant un polynôme en fonction de la taille de la donnée d'entrée n. Un algorithme est dit **exponentiel**, par convention, s'il n'est pas polynomial.

En se focalisant sur le facteur temporel, on distingue les classes de complexité suivantes :

Définition 1.41 (Classe P)
La classe **P** est l'ensemble des problèmes de décision pouvant être résolus en temps **P**olynomial par une machine de TURING déterministe.

Ce sont des problèmes relativement faciles c'est-à-dire ceux pour lesquels on connaît des algorithmes efficaces.

Définition 1.42 (Classe NP)
La classe **NP** est l'ensemble des problèmes pouvant être résolus en temps **P**olynomial par une machine de TURING **N**on déterministe.

En quelque sorte, c'est la classe des problèmes qui auraient été bien résolus si on disposait de la puissance considérable des machines non déterministes. Pour savoir si un problème donné appartient ou non à la classe NP, il suffit de pouvoir vérifier efficacement (en un temps polynomial) si une solution potentielle donnée est correcte. Le non déterminisme sert à 'masquer' le nombre exponentiel de solutions potentielles à tester. Par exemple, le problème consistant à trouver un cycle hamiltonien dans un graphe appartient à **NP** puisque, étant donné un cycle, il est trivial de vérifier en temps polynomial qu'il passe bien une et une seule fois chaque sommet.

Une machine de TURING déterministe est aussi une machine de Turing non déterministe d'où $\mathbf{P} \subseteq \mathbf{NP}$. En revanche, la réciproque $\mathbf{NP} \subseteq \mathbf{P}$, que l'on résume généralement à $\mathbf{NP} = \mathbf{P}$ du fait de la trivialité de l'autre inclusion, est l'un des problèmes ouverts les plus fondamentaux et les plus intéressant en informatique théorique. Cette question a été posée en 1970 pour la première fois et celui qui arrivera à prouver que **P** et **NP** sont différents ou égaux recevra le prix de Clay (plus de 1.000.000 US$).

Définition 1.43 (Classe CoNP)
La classe **CoNP** est l'ensemble des problèmes de décision dont les problèmes complémentaires appartiennent à la classe NP.
De même, on a $\mathbf{P} \subseteq \mathbf{CoNP}$.

On conjecture que $\mathbf{P} \neq \mathbf{NP}$ et que $\mathbf{NP} \neq \mathbf{CoNP}$.

Définition 1.44 (Réduction)
Un problème R est **réductible** en un problème P s'il existe une fonction g de complexité polynomiale telle que pour toute solution x de P, $y = g(x)$ soit une solution de R. On dit que le problème P est plus général que le problème R.

Définition 1.45 (Complétude)
On dit qu'un q'un problème *P* est complet pour sa classe si pour tout problème *R* de la même classe, *R* est réductible en *P*.

Définition 1.46 (Classe NP-complet, CoNP-complet)
Un problème est dit **NP-complet** s'il est complet pour la classe **NP**.
Un problème est dit **CoNP-complet** s'il est complet pour la classe **CoNP**.

Les problèmes **NP-complets** représentent les problèmes **NP** les plus difficiles à résoudre. Il est à noter que si un problème **NP-complet** admet un algorithme déterministe polynomial alors tout problème **NP** en admet un et donc **P = NP**. Le problème de satisfiabilité d'une formule propositionnelle, SAT en abrégé, est le problème **NP-complet** prototype. La classe des problèmes **NP-complets** n'a pas seulement un intérêt théorique. En effet de nombreux problèmes pratiques, relevant notamment du monde industriel (ordonnancement, étude de fiabilité, etc.) sont NP-complets. Les problèmes **NP-complets** sont très fréquents et nul n'a jamais réussi à trouver un algorithme déterministe polynomial résolvant l'un d'entre eux si bien qu'on conjecture que P ≠ NP.

Définition 1.47 (Classe NP-difficile)
Un problème est dit **NP-difficile** (ou **NP-dur**) si tout problème **NP** lui est réductible.
Donc ce n'est pas forcément un problème NP.

1.2.3. Hiérarchie polynomiale

On peut aller au-delà des classes NP et CoNP en se basant sur les deux conjectures NP ≠ CoNP et NP ≠ P. On définit alors la *hiérarchie polynomiale* (*HP*) qui repose sur la notion de *machine de Turing à oracle X*.

Définition 1.48 (Machine de TURING à oracle X)
Soit *X* une classe de complexité et Co*X* la classe des problèmes dont les problèmes complémentaires sont dans X. Une **machine de TURING** (déterministe ou non déterministe) **à oracle X** est une machine de TURING comportant un module spécial (l'oracle *X*) capable de résoudre n'importe quel problème de *X* ou de Co*X* en temps unité.

Définition 1.49 (P^X)
La classe $\mathbf{P}^{\mathbf{X}}$ contient les problèmes de décision pouvant être résolus en temps **p**olynomial par une machine de TURING déterministe à **oracle X**.

Définition 1.50 (NP^X)
La classe $\mathbf{NP}^{\mathbf{X}}$ contient les problèmes de décision pouvant être résolus en temps **p**olynomial par une machine de TURING **n**on déterministe à **oracle X**.

Définition 1.51 (Hiérarchie polynomiale)
On pose :

$$\Delta_0^{\mathbf{P}} = \Sigma_0^{\mathbf{P}} = \Pi_0^{\mathbf{P}} = \mathrm{P}$$

Pour tout $i \geq 0$,

$$\Delta_{i+1}^{\mathbf{P}} = \mathrm{P}^{\Sigma_i^{\mathbf{P}}}$$

$$\Sigma_{i+1}^{\mathbf{P}} = \mathbf{NP}^{\Sigma_i^{\mathbf{P}}}$$

$$\Pi_{i+1}^{\mathbf{P}} = \mathbf{co\text{-}NP}^{\Sigma_i^{\mathbf{P}}}$$

En particulier,

$$\mathbf{NP} = \Sigma_1^{\mathbf{P}} \text{ et } \mathbf{co\text{-}NP} = \Pi_1^{\mathbf{P}}$$

En plus

$$\Delta_2 \mathrm{P} = \mathrm{P}^{\mathbf{NP}}$$

La **hiérarchie polynomiale** (**HP**) est donc donnée par :

$$\mathbf{HP} = \bigcup_{i \geq 0} \Sigma_i^{\mathbf{P}}$$

La définition précédente implique les inclusions suivantes :

$$\Sigma_i^{\mathbf{P}} \subseteq \Delta_{i+1}^{\mathbf{P}} \subseteq \Sigma_{i+1}^{\mathbf{P}},$$

$$\Pi_i^{\mathbf{P}} \subseteq \Delta_{i+1}^{\mathbf{P}} \subseteq \Pi_{i+1}^{\mathbf{P}}.$$

Le fait de savoir si ces inclusions sont strictes reste une question ouverte, même si la plupart des spécialistes de la question le conjecture. Si, pour un entier *k* on :

$$\Sigma_k^{\mathbf{P}} = \Sigma_{k+1}^{\mathbf{P}}$$

Alors la hiérarchie s'effondre au niveau *k*. En particulier si **P = NP** la hiérarchie s'effondre complètement. Mais il est fortement conjecturé que la hiérarchie polynomiale HP ne s'effondre pas ; c'est une hiérarchie infinie.

Définition 1.52 (Classe $\Delta_2^{\mathbf{P}}$ [*O*(log *n*)])

La classe $\Delta_2^{\mathbf{P}}$ **[*O*(log *n*)]** est le sous ensemble $\Delta_2^{\mathbf{P}} = \mathrm{P}^{\mathbf{NP}}$ obtenu en exigeant que le nombre d'oracles soit borné par une fonction logarithmique de la taille de la donnée d'entrée.

La figure 1.1 est une représentation graphique de la hiérarchie polynomiale.

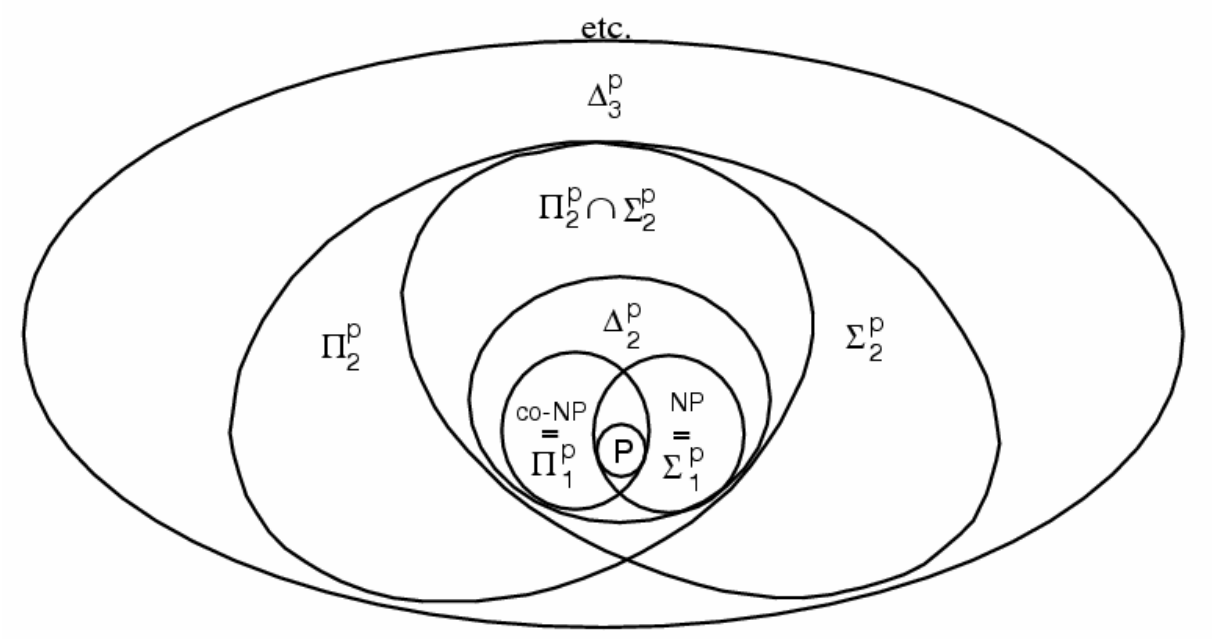


Figure 1.1. Hiérarchie polynomiale.

La classification des problèmes dans la classe $\mathbf{\Delta_2^P}$ peut être à son tour raffinée et ce en définissant la *hiérarchie booléenne HB*.

Définition 1.53 (Hiérarchie booléenne)
On pose :
$\mathbf{BH_0} = \mathbf{P}$
$\mathbf{BH_1} = \mathbf{NP}$
$\mathbf{BH_2} = \{P_1 \cap P_2 : P_1 \in \mathbf{NP}$ et $P_2 \in \mathbf{CoNP}\}$ et donc $\mathbf{CoBH_2} = \{P_1 \cup P_2 : P_1 \in$ NP et $P_2 \in$ CoNP$\}$
$\mathbf{BH_{2k+1}} = \{P_1 \cup P_2 : P_1 \in \mathbf{NP}$ et $P_2 \in \mathbf{BH_{2k}}\}$ $k \geq 1$
$\mathbf{BH_{2k+2}} = \{P_1 \cap P_2 : P_1 \in$ NP et $P_2 \in \mathbf{CoBH_{2k+1}}\}$ $k \geq 1$
La **hiérarchie booléenne** (**HB**) est alors définie par :
$\mathbf{HB} = \cup\, \mathbf{BH_k}$ ($k \geq 0$).

$\mathbf{BH_2}$ est dénoté aussi par **DP**.

Pour plus de détails quant à la théorie de la complexité, le lecteur pourra se référer utilement aux ouvrages de Garey et Johnson [Garey and Johnson 1979] et Papadimitriou [Papadimitriou 1994].

Chapitre 2

Compilation de bases de connaissances propositionnelles

Sommaire

« Si nous donnons un poisson à homme, il n'aura pas faim pendant une journée, mais si nous lui apprenons à pêcher, il n'aura pas faim toute sa vie. »
Proverbe africain.

2.1. Introduction à la compilation de connaissances

La compilation est une technique assez fréquente en informatique. Souvent, les compilateurs des langages de programmation effectuent un prétraitement sur le code objet afin de l'optimiser. Un autre exemple peut être le prétraitement d'un graphe pour obtenir une structure de données qui accélère certaines opérations. Alors que cette compilation tourne autour de problèmes qui sont déjà résolus en temps polynomial, la compilation de connaissances est caractérisée par le fait que les problèmes de raisonnement en Intelligence Artificielle sont soit NP-complets soit coNP-complets (ou même plus difficiles). Une définition de la compilation en général et de la compilation de connaissances en particulier a été donnée dans [Ghallab 1988] comme suit :

Définition 2.1 (Compilation)
On entend par **compilation** une transformation d'une représentation d'entrée vers une représentation de sortie. Cette transformation comporte une réécriture en un langage approprié (par exemple passage d'un algorithme écrit en langage de haut niveau vers le même algorithme écrit en langage machine). Eventuellement, la compilation s'accompagne d'une réorganisation, globale ou locale, des informations représentées en des structures de données particulièrement adaptées à certains traitements. Plus précisément, par **compilation de bases de connaissances**, on entend tout prétraitement qui réduit l'ordre de complexité des algorithmes mis en œuvres dans l'exploitation de la base.

La clef en compilation de connaissances est qu'un problème est conceptuellement divisé en deux parties : une base de connaissances et une requête. Tandis que la requête est variable (elle change entre deux instances du problème), la base de connaissances est rarement modifiée et la même base est utilisée afin de répondre à un bon nombre de requêtes. Le principe de la compilation de connaissances se résume comme suit :

- Dans un premier temps, on effectue un prétraitement sur la base de connaissances qui est une partie qu'on qualifie généralement de fixe ce qui produit une structure de données appropriée. Cette phase est dite *raisonnement off-line.*
- Ensuite, on répond à la requête en utilisant le résultat de la phase off-line. Cette deuxième phase est dite *raisonnement on- line.*

L'idée de fond est de déplacer la surcharge de travail de la phase on-line vers la phase off-line de façon à alléger les traitements nécessaires pour obtenir la réponse à une requête. Etant donné que la phase off-line n'est appelée qu'une seule fois, ce coût sera amorti par les multiples phases on-line (qui elles, se trouvent allégées). Un peu comme si une bibliothèque fermait quelques jours pour ranger ses livres de façon à pouvoir, à l'issue du rangement, trouver plus vite les livres demandés.

Exemple 2.1
Soit le problème suivant [$\Sigma \models l, \Sigma, l$] défini par sa requête $\Sigma \models l$, sa partie fixe Σ et sa partie variable l où Σ représente une formule sous forme CNF et l un littéral définis sur un ensemble fini de variables propositionnelles V. Il s'agit d'un problème CoNP-complet. Le but de la compilation est de rendre le raisonnement on-line plus efficace. Au fait, ceci est possible en enregistrant sur une table, pour chaque littéral l apparaissant dans Σ si $\Sigma \models l$ ou non. La taille de cette table est en $O(n)$ où n est la taille de Σ (le nombre d'atomes distincts qu'elle contient). En outre, cette table peut être consultée en un temps

$O(n)$. Construire la table revient à résoudre $O(n)$ instances de problèmes CoNP-complets. Bien que la phase off-line requiert un temps considérable, l'essentiel est qu'il soit fini.

La figure 2.1 illustre le procédé de compilation de connaissances. Dans cette figure, on peut distinguer la partie fixe f (base de connaissances) et la partie variable v (la requête). Lors de la phase off-line, f est compilée en une autre entité D_f. Quant à la phase on-line, elle est représentée par l'algorithme ASK.

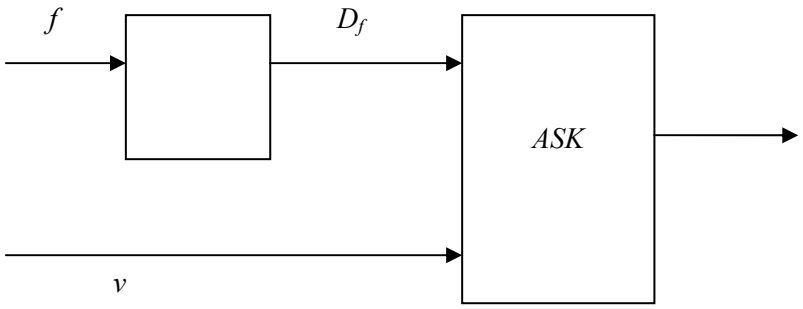


Figure 2.1. Compilation d'une base de connaissances.

Définition 2.2 (Problème compilable)
Etant donné un problème défini par le triplet [P, F, V] où P désigne la requête à laquelle on veut répondre, F sa partie fixe et V sa partie variable. Ce problème est dit **compilable** s'il existe deux polynômes p_1, p_2 et un algorithme ASK tel que pour toute instance f de F, il existe une structure de données D_f vérifiant :

1. $| D_f | \leq p_1(| f |)$;
2. pour toute instance v de V, l'appel ASK(D_f, v) retourne 'OUI' si et seulement si $<f, v>$ est une instance 'OUI' de P ;
3. ASK(D_f, v) requiert un temps $\leq p_2(| v | + | D_f |)$ [Cadoli *et al.* 1996].

Autrement dit :

1. la structure de données D_f engendrée par la phase off-line doit occuper un espace polynomial par rapport à la taille de la partie fixe;
2. l'algorithme de la phase on-line ASK doit être correct et complet ;
3. ASK doit de surcroît avoir un temps d'exécution qui soit polynomial par rapport à la taille de ses entrées.

L'explication du premier point est très simple. Sans cette restriction, il serait physiquement impossible de stocker la base compilée sur une machine. Le second point stipule que toutes les requêtes qui ont une réponse positive dans le problème initial, doivent aussi avoir une réponse positive dans le problème compilé (cela correspond à un raisonnement correct) et seules les réponses qui ont une réponse positive dans le problème initial ont une réponse positive dans le problème compilé (raisonnement complet). Enfin, le dernier point permet de s'assurer que la version compilée du problème est bien dans une classe de complexité inférieure à celle du problème initial. Si seul le

second point n'est pas respecté (dans une certaine mesure) alors on se retrouve plutôt dans le cadre d'une compilation *approximative*.

Exemple 2.2
Selon cette définition, le problème [Σ |= *l*, Σ, *l*] de l'exemple 2.1 est compilable.

Il a à noter que si le nombre des requêtes possibles est d'une taille polynomiale par rapport à la taille de la partie fixe alors le problème en question est compilable. Il s'agit d'une condition suffisante; en effet il existe des problèmes compilables qui ne vérifient pas cette condition.

Exemple 2.3
Etant donné le problème [Σ |= *t*, Σ, *t*] où *t* est un terme. Le nombre de termes possibles est au moins de 2^n (*n* étant la taille de l'ensemble des variables propositionnelles sur lequel sont définis Σ et *t*). Néanmoins, la table de l'exemple 2.1 est suffisante pour avoir un raisonnement on-line polynomial. En effet, pour tout terme $t = l_1 \wedge \ldots \wedge l_m$, Σ |= *t* si et seulement si $(\Sigma \models l_1) \wedge \ldots \wedge (\Sigma \models l_m)$.

En revanche, on ignore jusqu'à présent si le problème [Σ |= *c*, Σ, *c*] où *c* est une clause, est compilable ou non. Au fait, il a été démontré dans [Cadoli *et al*. 1996] que si [Σ |= *c*, Σ, *c*] est compilable alors $\Sigma_2^P = \Pi_2^P$ (ce qui est conjecturé faux). En outre, il est facilement démontrable que si le problème [Σ |= *f*, Σ, *f*] où *f* est une formule propositionnelle quelconque est compilable alors P = NP.

On fait remarquer que la définition d'un problèmes non compilable découle directement de la définition d'un problèmes compilable : un problème [P, F, V] est dit non compilable si un des polynômes p_1 ou p_2 n'existe pas. Une condition nécessaire (non suffisante) pour qu'un problème soit non compilable est que la taille de l'ensemble de ses requêtes possibles soit exponentielle.

Quant aux méthodes de compilation, elles peuvent être classifiées en deux grandes familles. La première famille comprend les méthodes de compilation qui préservent l'équivalence, c'est-à-dire celles qui génèrent un résultat équivalent à la base initiale. Ces méthodes sont dites méthodes de compilation *exactes*. La seconde famille regroupe les méthodes de compilation approximatives qui produisent un résultat qui n'est pas équivalent à la base initiale et donc ne permettent d'effectuer qu'une partie des déductions pouvant être faites depuis la base initiale.

2.2. Méthodes classiques de compilation exacte

En compilation exacte, on peut citer trois principales méthodes classiques :

- Compilation basée impliquants premiers ou impliqués premiers;
- Compilation par complétude de la résolution unitaire.
- Compilation basée impliqués premiers modulo une théorie traitable.

2.2.1. Compilation basée impliquants premiers ou impliqués premiers

On rappelle qu'un impliquant d'une base de connaissance Σ est un terme fondamental (conjonction de littéraux qui ne contient pas de littéraux complémentaires) *T* tel que *T* |= Σ. Chaque modèle de la

base de connaissances correspond à un impliquant dans lequel chaque variable apparaît comme un littéral positif si elle est affectée la valeur VRIA, et comme littéral négatif si elle est affectée la valeur FAUX dans le modèle. De plus T est dit impliquant premier de Σ si pour tout autre impliquant T', Lit(T') $\not\subset$ Lit(T). D'autre part, un impliqué de cette base est une clause fondamentale C (disjonction de littéraux qui ne contient pas de littéraux complémentaires) tel que Σ |= C. En outre, C est dit impliqué premier si pour tout autre impliqué C', Lit(C') $\not\subset$ Lit(C).

La disjonction de tous les impliquants premiers $T_1,\ldots,T_k$ d'une base de connaissances Σ génère une formule DNF IP(Σ) = $T_1 \vee \ldots \vee T_k$ qui est équivalente à Σ et tel que pour toute requête R, Σ |= R si et seulement si pour tout impliquant premier T_i, T_i |= R. Si R est une formule CNF alors ça revient à vérifier que chaque clause de R a une intersection non vide avec chaque impliquant premier de Σ. Par conséquent, la déduction des requêtes sous forme CNF s'effectue en temps polynomial par rapport à la taille de la formule IP(Σ) des impliquants premiers plus la taille de la requête.

En revanche, la conjonction de tous les impliqués premiers $C_1,\ldots, C_k$ d'une base de connaissances Σ engendre une formule CNF PI(Σ) = $C_1 \wedge \ldots \wedge C_k$ qui est équivalente à Σ. En plus, pour toute requête R sous forme CNF, Σ |= R si et seulement si pour chaque clause fondamentale C' de R, il existe un impliqué premier C tel que C |= C' c'est-à-dire Lit(C) $\subseteq$ Lit(C') si les clauses sont vues comme étant des ensembles de littéraux. Par conséquent, la déduction des requêtes sous forme CNF s'effectue en temps polynomial par rapport à la taille de la formule PI(Σ) des impliqués premiers plus la taille de la requête. Il est à noter que la méthode d'impliqués premiers est la plus utilisée en compilation de connaissances.

Exemple 2.4
Soit Σ = $\{a \Rightarrow b, b \Rightarrow c\}$.
PI(Σ) = $(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee c)$

Intuitivement, les impliqués premiers d'une base de connaissances sous forme CNF peuvent être générés en résolvant les clauses de la base de connaissances (chaque résolvante est un impliqué) ensuite éliminer ceux qui ne sont pas premiers. Néanmoins, cette méthode requiert un nombre prohibitif de résolutions. Le problème de calcul des impliqués premiers d'une formule CNF est apparu il y'a longtemps et il a fait l'objet de nombreuses études [Tison 1967, Slagle *et al*. 1970, De Kleer 1992]

D'après [Chandra and Markowsky 1978], le nombre d'impliquants premiers et d'impliqués premiers d'une base de connaissances ayant n atomes est exponentiel en n. Dans [Schrag and Crawford 1996] on retrouve une étude expérimentale concernant le nombre d'impliqués et d'impliqués premiers en fonction du rapport (nombre de clauses)/(nombre de variables). Ce rapport a été étudié dans de nombreuses expérimentations relatives au problème de satisfiabilité. Pour une base de connaissances dont ce rapport est autour d'une valeur critique de 4.3, le nombre d'impliqués premiers est exponentiel par rapport à sa taille. Donc, on propose d'utiliser de telles bases de connaissances comme étant des benchmarks afin de tester les méthodes de compilation de connaissances basées impliqués premiers.

La plupart des méthodes dédiées à la génération des impliqués premiers peuvent être adaptées pour la génération des impliquants premiers. Par ailleurs, dans [Schrag 1996], on propose une méthode spécifique à la génération d'impliquants premiers basée sur une variante de la fameuse procédure Davis & Putnam destinée à la résolution du problème SAT; si la formule est satisfiable alors elle retourne une assignation de vérité aux différents atomes satisfaisant la formule considérée. Comme la variante en question est exhaustive dans le sens où elle explore tout l'espace de recherche et ne s'arrête pas dès que la formule est satisfaite pour la première fois, elle produit tous les impliquants à partir desquels les impliquants premiers seront sélectionnés.

2.2.2. Compilation par complétude de la résolution unitaire

Etant donné que le nombre d'impliqués premiers est en général exponentiel, del Val [del Val 1994] propose une nouvelle méthode de compilation basée impliqués premiers qui s'avère plus efficace. On a vu précédemment que la déduction d'une requête CNF à partir d'une base de connaissances compilée sous forme de conjonction d'impliqués premiers se fait en vérifiant pour chaque clause de la requête, s'il existe un impliqué premier dont elle est conséquence logique c'est-à-dire un impliqué premier qu'elle inclut (si une clause est vue comme étant un ensemble de littéraux). D'autre part, ce test d'inclusion peut être réalisé via une réfutation par résolution unitaire (une forme de résolution où une des deux clauses résolvantes est une clause unitaire). La résolution unitaire est saine mais incomplète dans le sens où il existe des réfutations qu'elle n'arrive pas à déceler. Nier une clause dans la requête produit un ensemble de littéraux, et via le résolution unitaire, on obtient la clause vide si et seulement s'il existe un impliqué premier qui est un sous ensemble de l'ensemble des littéraux de la clause niée.

Toutefois, la résolution unitaire peut faire plus qu'un simple test d'inclusion. En effet, grâce à la résolution unitaire, on a pas besoin de garder tous les impliqués premiers, mais seulement un sous ensemble de ces derniers à partir duquel tout autre impliqué peut être calculé par résolution unitaire.

del Val [del Val 1994] suggère d'utiliser n'importe quel algorithme qui génère des impliqués premiers, ensuite éliminer tous les impliqués premiers qui peuvent être reconstruits par résolution unitaire durant la phase on-line. On rappelle que la plupart des algorithmes calculant les impliqués premiers d'une base de connaissances opèrent en effectuant des résolutions répétitives. En effet, la méthode de del Val garde uniquement les résolvantes dite *fusion*.

Définition 2.3 (Résolvante fusion)
Une résolvante de deux clauses c_1 et c_2 est dite **résolvante fusion** si après avoir effectué une résolution sur un littéral l, la résolvante contient deux littéraux identiques c'est-à-dire $\text{Lit}(c_1)-\{l\} \cap \text{Lit}(c_2)-\{\neg l\} \neq \varnothing$.

Afin de voir l'utilité de garder les résolvantes fusion, on donne l'exemple suivant :

Exemple 2.5
On suppose que la base de connaissances contienne les deux clauses:
$C_1 = a \vee b \vee \neg l$
$C_2 = b \vee c \vee l$

Leur résolvante est :
$C_3 = a \vee b \vee c$
Si l'on considère les clauses unitaires $\neg a$ et $\neg c$, à partir des clauses C_1, C_2 et C_3, on peut dériver b par résolution unitaire. En éliminant C_3, la résolution unitaire génère les deux clauses $b \vee \neg l$ et $b \vee l$ qui ne peuvent pas être résolues via la résolution unitaire et donc b n'est pas dérivable. Donc la clause C_3 qu'est une résolvante fusion n'est pas redondante par rapport à la résolution unitaire et doit être gardée dans le processus de compilation.

del Val donne des cas où la compilation par complétude de la résolution unitaire peut éliminer un nombre exponentiel d'impliqués premiers. En outre, il montre que l'adaptation de l'algorithme de Tison [Tison 1967] dédié au calcul des impliqués premiers réduit considérablement le nombre d'impliqués premiers non redondants et donc réduit la taille de la base compilée.

2.2.3. Compilation basée impliqués premiers modulo une théorie

Probablement, jusqu'à présent, la méthode de compilation basée impliqués premiers la plus raffinée est celle proposée par Pierre Marquis [Marquis 1995]. Dans un premier temps, il remarque que les méthodes basées impliqués premiers / impliquants premiers transforment le problème $\Sigma \models R$ (R étant une formule CNF), qui implique la base de connaissances entière, en des tests locaux impliquant un seul impliqué/impliquant premier à la fois. Il propose alors d'améliorer ces tests locaux en utilisant une théorie.

La compilation basée impliqués premiers modulo une théorie se base sur le principe bien connu de '*diviser pour mieux régner*'. En effet, partant du constat qu'une base de connaissances Σ peut souvent être scindée en deux parties distinctes : une '*traitable*' Φ et une '*difficile*' Ψ telles que $\Phi \wedge \Psi \equiv \Sigma$, Marquis propose de restreindre le calcul des impliqués de Σ au calcul des *impliqués de* Σ *modulo* Φ. L'idée est d'utiliser autant que possible la partie facile dans le traitement de la partie difficile.

Définition 2.4 (Impliqué modulo une théorie)
Soit Φ une théorie. Un **impliqué modulo Φ** d'une base de connaissances Σ est une clause C tel que $\Sigma \cup \Phi \models C$ que l'on note également par $\Sigma \models_\Phi C$.

Définition 2.5 (Impliqué premier modulo une théorie)
Soit Φ une théorie. Un **impliqué premier modulo Φ** d'une base de connaissances Σ est un impliqué C modulo Φ de Σ tel que pour tout autre un impliqué C' modulo Φ de Σ on a : si $C' \models_\Phi C$ alors $C \models_\Phi C'$.

On remarque que si l'on considère la théorie vide ($\Phi = \varnothing$), on retrouve les définitions usuelles d'impliqués et d'impliqués premiers. D'autre part, étant donnée une théorie Φ, pour toute requête R sous forme CNF, $\Sigma \models R$ si et seulement pour chaque clause C' de R, il existe C : un impliqué premier modulo Φ de Σ tel que $C \models_\Phi C'$. Il est de même, en considérant la théorie vide, on retrouve la propriété des impliqués premiers.

On fait remarquer que vérifier $C \models_\Phi C'$ est équivalent à vérifier pour chaque littéral $l_i \in C$, si $\Phi \models \neg l_i \vee C$'. L'idée est que si la déduction dans Φ se fait en temps polynomial alors la déduction des

requêtes CNF se fait aussi en temps polynomial par rapport à la taille de l'ensemble des impliqués premiers modulo une théorie. Marquis suggère de prendre comme théorie l'ensemble de toutes les clauses de Horn de Σ, de toutes ses clauses binaires et en général n'importe quel sous-ensemble de Σ qui permet une déduction polynomiale.

2.3. Méthodes classiques de compilation approximative

On a vu précédemment, que certains problèmes sont compilables, tandis que pour d'autres, on ignore jusqu'à présent s'ils le sont ou non. Afin de remédier à ce problème, on propose une compilation approximative. La notion de l'approximation réside dans le fait qu'il faut changer quelque chose afin de pouvoir traiter le problème aux dépens de la l'équivalence logiques entre la base initiale et la base compilée.

L'idée de la compilation approximative est de changer l'espace des solutions du problème initial. C'est-à-dire que certaines instances qui étaient rejetées dans le problème initial vont se trouver acceptées, et vice vers ça. Pour s'y retrouver, l'idée sous-jacente est en fait de réaliser une première transformation qui ajoutera des instances à l'espace des solutions initial (et n'en enlèvera pas), et une autre qui ne fera que restreindre l'espace des solutions initial (et n'ajoutera pas de solutions).

Avec la première transformation, on obtient une borne supérieure puisque l'espace des solutions à compiler est strictement inclus dans l'espace transformé. Une réponse positive de la borne supérieure n'est pas une réponse sûre (on ne peut rien dire) puisque la transformation a ajouté des solutions. En revanche, si la réponse est négative, puisque aucune des solutions initiales n'a été supprimée, on peut en déduire que la réponse du problème initial est négative aussi.

Avec la seconde transformation, on obtient une borne inférieure. L'espace des solutions de la borne est strictement inclus dans l'espace des solutions à compiler. Le comportement de cette borne est dual à celui de la borne supérieure et donc ce sont les réponses positives de cette borne qui sont sûres.

La figure 2.2 illustre le comportement de la compilation approximative. La forme en noir représente l'espace des solutions du problème à compiler. Si le point est dans la forme noire, alors le point est une instance positive, sinon, c'est une instance négative. Le rectangle grisé est une borne supérieure puisque tout point se trouvant dans la forme noire, se trouve aussi dans le rectangle grisé. Le rectangle blanc est une borne inférieure puisque tout point du rectangle blanc est inclus dans la forme noire. Cependant, les deux bornes ne permettent pas de répondre à toutes les requêtes, et donc on aura une zone d'incertitude (zone qui est hachurée dans la figure 2.2). Naturellement, plus la zone d'incertitude est petite, meilleure est la compilation approximative.

Formellement, on définit les bornes supérieures et inférieures comme suit :

Définition 2.6 (Borne supérieure)
Une approximation A d'une base de connaissances Σ est dite **saine** si pour toute requête R, si $A \models R$ alors $\Sigma \models R$. A est dite **borne supérieure** (ou **UB** pour Upper Bound) de Σ.

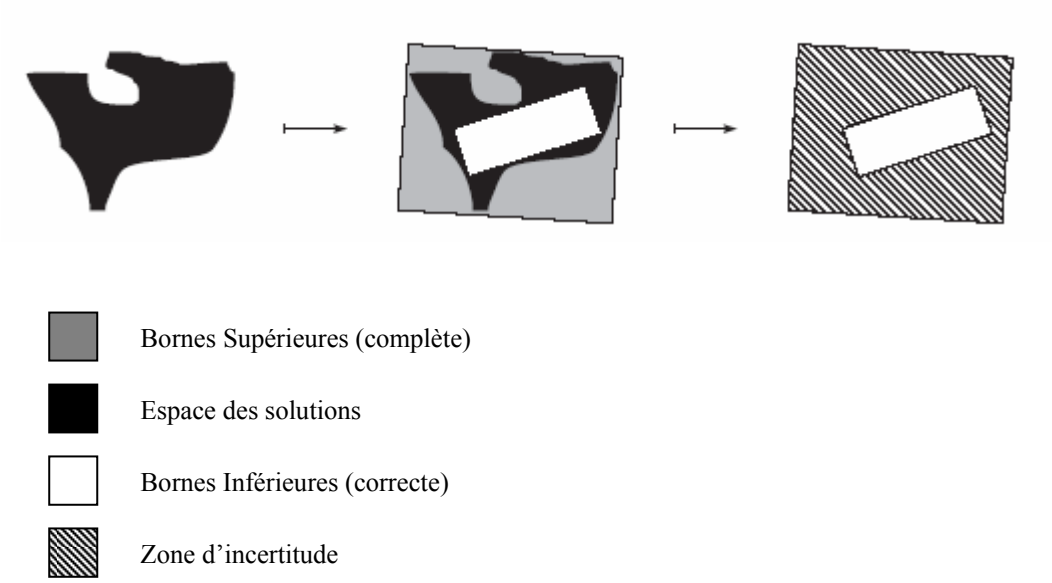


Figure 2.2. Compilation approximative.

On note que A est une UB de Σ si et seulement si $\Sigma \models A$. En fait, pour le premier sens de cette équivalence, il suffit de prendre dans la définition $R = A$: on sait que $A \models A$ et donc on déduit que $\Sigma \models A$. Pour l'autre sens, on suppose que $\Sigma \models A$ donc pour toute requête R, si $A \models R$ alors $\Sigma \models A$ par transitivité de la relation de déduction. D'une manière duale :

Définition 2.7 (Borne inférieure)
Une approximation B de Σ est dite **complète** si pour toute requête R, si $B \not\models R$ alors $\Sigma \not\models R$. B est dite **borne inférieure** (ou **LB** pour Lower Bound) de Σ.

En plus, on remarque que B est une LB de Σ si et seulement si $B \models \Sigma$.

Dans la compilation par approximation, il est crucial d'avoir des bornes qui soient le plus proche possible du problème à compiler. Sans quoi, avec des bornes trop grossières, on pourrait se trouver dans les cas extrêmes où la borne supérieure englobe toutes les instances possibles et la borne inférieure rejette systématiquement toutes les instances. De telles approximations n'apportent aucune information. Sans tomber dans cet extrême, il est possible d'obtenir des bornes supérieures qui sont strictement incluses dans une autre borne supérieure; et dans ce cas c'est la borne la plus petite qui est la meilleure. Cette dernière est dite **LUB** (pour Least Upper Bound). Pour les bornes inférieures, on fait les considérations duales aux précédentes : ce sont les bornes les plus grosses qui sont les plus intéressantes. La borne inférieure la plus grosse est dite **GLB** (pour Greatest Lower Bound). D'ailleurs cela apparaît clairement sur la figure 2.2 : plus le rectangle blanc est grand et le gris est petit, plus la zone d'incertitude s'affine et meilleure est l'approximation.

Définition 2.8 (GLB, LUB)
Formellement, une **GLB** d'une base de connaissances Σ notée Σ_{GLB} est définie comme étant une LB de Σ, et tel que pour toute autre formule Σ', si $(\Sigma_{GLB} \models \Sigma' \models \Sigma)$ alors $(\Sigma' \models \Sigma_{GLB})$. D'autre part, on définit une **LUB** d'une base de connaissances Σ notée Σ_{LUB} comme étant une UB de Σ tel que pour toute autre formule Σ', si $(\Sigma \models \Sigma' \models \Sigma_{LUB})$ alors $(\Sigma_{LUB} \models \Sigma')$.

2.3.1. Versions anytime des méthodes exactes

Les méthodes de compilation exacte décrites plus haut peuvent être arrêtées prématurément à n'importe quel moment en fournissant une approximation de la base de connaissances. Plus précisément, les méthodes basées impliquants génèrent dans une LB: si une requête *R* n'est pas impliquée par un des impliquants déjà calculés d'une base Σ, alors R n'est pas impliquée par Σ. D'autre part, les méthodes basées impliqués génèrent une UB : si l'un des impliqués déjà calculés implique la requête *R*, alors *R* est aussi impliquée par Σ.

2.3.2. Approximations de Horn

Dans [Selman and Kautz 1991], on propose une méthode originale de compilation approximative. L'idée de base de cette méthode est de compiler une base de connaissances vers une formule appartenant à une classe syntaxique qui garantit une inférence en temps polynomial. Par exemple, une base Σ peut être compilée en une formule de Horn (un ensemble de clauses de Horn) Σ'étant donné que $\Sigma' \models c$ où c est une clause peut être résolu en un temps $O(|\Sigma'| + |c|)$.

Exemple 2.6
Soit Σ la base de connaissances suivante :
$\Sigma = (a \Rightarrow b) \wedge (c \Rightarrow b) \wedge (a \vee c)$.

Chacune des deux formules $\Sigma_{LB1} = a \wedge b \wedge c$ et $\Sigma_{LB2} = a \wedge b$ sont des LBs de Horn de Σ. Cependant, Σ_{LB2} est une approximation meilleure que Σ_{LB1} étant donné que $\Sigma_{LB1} \models \Sigma_{LB2} \models \Sigma$. Dans ce cas, Σ_{LB2} est une GLB de Horn de Σ.

En revanche, les deux formules $\Sigma_{UB1} = (a \Rightarrow b) \wedge (c \Rightarrow b)$ et $\Sigma_{UB2} = b$ sont des UBs de Σ. En outre, $\Sigma \models \Sigma_{UB2} \models \Sigma_{UB1}$ si bien que l'approximation Σ_{UB2} soit meilleure que Σ_{UB1}. Σ_{UB2} est une LUB de Σ.

2.4. Langages cibles de compilation

Les méthodes classiques de compilation de connaissances (entre autres celles qu'on a esquissées précédemment) ont pour langage cible des variantes de CNF ou de DNF. En outre, ces méthodes considèrent souvent un seul type de requêtes : la déduction clausale c'est-à-dire vérifier si une clause est conséquence logique d'une base de connaissances ou non.

Dans [Darwiche and Marquis 2002], on présente un nombre considérable de langages cibles de compilation de connaissances issus de travaux en intelligence artificielle, en vérification formelle et en informatique de manière générale. En outre, on met en évidence trois facteurs permettant de comparer ces langages et surtout de choisir le langage le plus commode pour une application particulière. En fait, pour un langage donné, on s'intéresse à sa *concision*, à la classe des requêtes auxquelles il permet de répondre en temps polynomial ainsi qu'à la classe des transformations qu'il permet d'appliquer en temps polynomial aussi. Donc étant donnée une application, on identifie en premier lieu l'ensemble des requêtes et des transformations auxquelles elle fait appel, ensuite on opte pour le langage le plus concis qui supporte ces opérations en temps polynomial.

2.4.1. Langages dérivés du langage NNF

Les langages cibles considérés sont tous des dérivés d'un même langage dit NNF défini comme suit :

Définition 2.9 (NNF)
Une formule propositionnelle est sous forme **NNF** (pour Negation Normal Form) si elle est construite à partir de littéraux en utilisant uniquement les opérateurs de conjonction ($\wedge$) et de disjonction ($\vee$) [Barwise 1977].

Les graphes dirigés acycliques ou DAG (pour Directed Acyclic Graphs) constituent une bonne représentation graphiques des formules NNF où chaque feuille du DAG est étiquetée par $\bot$, $\top$ ou un littéral et chaque nœud interne est étiqueté par une conjonction '$\wedge$' ou par une disjonction '$\vee$'. La taille d'une formule NNF est donnée par le nombre d'arcs dans le DAG correspondant. Quant à sa hauteur, elle est donnée par le nombre maximum d'arcs entre la racine et une feuille du DAG.

Exemple 2.7
La figure 2.3 représente une formule NNF (la fonction de parité). Les fils de chaque nœud figurent en dessous de lui si bien qu'on n'ait pas besoin de montrer l'orientation des arcs du DAG.

Toute formule propositionnelle peut être représentée par une formule NNF donc le langage NNF est complet. Il importe de faire la différence entre un langage de représentation de connaissances et un langage cible de compilation de connaissances. Alors que le premier doit être assez naturel afin de permettre de coder des connaissances, le second quant à lui, doit permettre de raisonner en temps polynomial. En effet, plus le langage cible de compilation est puissant, plus il est loin de permettre à l'homme de spécifier ou d'interpréter directement des connaissances.

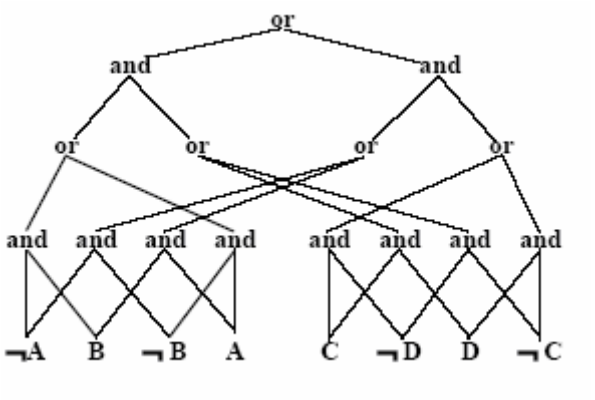


Figure 2.3. Formule sous forme NNF.

Définition 2.10 (Langage cible de compilation)
Un langage *L* est qualifié de **langage cible de compilation** s'il permet au moins la déduction clausale en temps polynomial.

Le langage NNF n'est pas qualifié de langage cible de compilation (à moins que P = NP) [Papadimitriou 1994]. Cependant, un bon nombre de ses sous ensembles l'est. Les sous ensembles du

langage NNF sont définis en imposant certaines restrictions sur ce dernier. Da manière générale, on distingue deux sous ensembles : les langages *plats* et les langages *imbriqués*.

Les propriétés nous permettant de définir les langages plats sont les suivantes :

Définition 2.11 (Aplatissement)
Soit L un langage NNF. L vérifie la propriété d**'aplatissement** si et seulement si la hauteur de toute formule de L est au maximum de 2.

Définition 2.12 (Disjonction simple)
Soit L un langage NNF. L vérifie la propriété de **disjonction simple** si et seulement si les fils de tout nœud de disjonction ($\vee$) sont des feuilles qui ne partagent pas de variables (le nœud est une clause fondamentale).

Définition 2.13 (Conjonction simple)
Soit L un langage NNF. L vérifie la propriété de **conjonction simple** si et seulement si les fils de tout nœud de conjonction ($\wedge$) sont des feuilles qui ne partagent pas de variables (le nœud est un terme fondamental).

Définition 2.14 (f-NNF)
Le langage **f-NNF** est le sous-ensemble du langage NNF qui satisfait la propriété d'aplatissement.

On note que le langage **CNF** est le sous-ensemble du langage f-NNF satisfaisant la propriété de disjonction simple tandis que le langage **DNF** est le sous-ensemble de f-NNF satisfaisant la propriété de conjonction simple. CNF ne permet pas une déduction clausale en temps polynomiale si bien qu'il n'est pas qualifié de langage cible de compilation. En revanche son dual DNF l'est. Les langages **PI** des impliqués premiers et **IP** des impliquants premiers sont des sous ensembles de CNF et DNF respectivement.

Maintenant, on considère les sous ensembles **imbriqués** du langage NNF. Ces langages sont définis via les propriétés suivantes : *décomposabilité*, *déterminisme*, *régularité*, *décision* et *ordre*.

Définition 2.15 (Décomposabilité)
Soit $C = C_1 \wedge \ldots \wedge C_n$ un nœud de conjonction '$\wedge$' d'une formule NNF. C est dit **décomposable** si et seulement si ses fils ne partagent pas de variables c'est-à-dire $\forall\, i, j \in \{1, \ldots, n\}$ avec $i \neq j$: $\mathrm{Var}(C_i) \cap \mathrm{Var}(C_j) = \varnothing$. Une formule NNF satisfait la propriété de **décomposabilité** si et seulement si tous ses nœuds de conjonction sont décomposables. [Darwiche 2001a].

Définition 2.16 (Déterminisme)
Soit $D = D_1 \vee \ldots \vee D_n$ un nœud de disjonction '$\vee$' d'une formule NNF. D est dit **déterministe** si et seulement si ses fils sont deux à deux logiquement contradictoires c'est-à-dire $\forall\, i, j \in \{1, \ldots, n\}$ avec $i \neq j : D_1 \wedge D_j \models \bot$. Une formule NNF satisfait la propriété de **déterminisme** si et seulement si tous ses nœuds de disjonction sont déterministes. [Darwiche 2001b]

Définition 2.17 (Uniformité)
Soit $D = D_1 \vee \ldots \vee D_n$ un nœud de disjonction '$\vee$' d'une formule NNF. D est dit **uniforme** si et seulement si tous ses fils partagent le même ensemble de variables c'est-à-dire $\forall i, j \in \{1, \ldots, n\}$ avec $i \neq j$: $\mathrm{Var}(D_i) = \mathrm{Var}(D_j)$. Une formule NNF satisfait la propriété d'**uniformité** si et seulement si tous ses nœuds de disjonction sont uniformes. [Darwiche 2001b]

En sa basant sur ces trois propriétés, on définit les langages suivants :

Définition 2.18 (DNNF)
Le langage **DNNF** est le sous-ensemble de NNF satisfaisant la propriété de décomposabilité.

Définition 2.19 (d-NNF)
Le langage **d-NNF** est le sous-ensemble de NNF satisfaisant la propriété de déterminisme.

Définition 2.20 (s-NNF)
Le langage **s-NNF** est le sous-ensemble de NNF satisfaisant la propriété d'uniformité.

Définition 2.21 (d-DNNF)
Le langage **d-DNNF** le sous ensemble de NNF satisfaisant et la décomposabilité et le déterminisme.

Définition 2.22 (sd-DNNF)
Le langage **sd-DNNF** est le sous-ensemble de NNF satisfaisant les propriétés de décomposabilité, de déterminisme et d'uniformité.

Exemple 2.8
Soit le nœud '$\wedge$'marqué dans la figure 2.4. Ce nœud possède deux fils. Le premier contient les variables A et B tandis que le second contient les variables C et D. Ce nœud de conjonction est alors décomposable. Tout autre nœud $\wedge$ dans cette figure est décomposable donc la formule NNF considérée est DNNF.

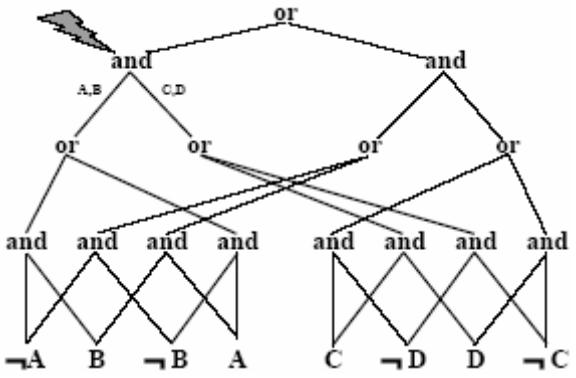


Figure 2.4. Décomposabilité.

Exemple 2.9
Soit le nœud '$\vee$' marqué dans la figure 2.4. Ce nœud a deux fils correspondant aux formules $\neg A \wedge B$ et $\neg B \wedge A$. Ces deux formules sont logiquement contradictoires donc ce nœud de disjonction est

déterministe. Il est de même pour les autres nœuds de disjonction si bien que cette formule NNF soit d-NNF.

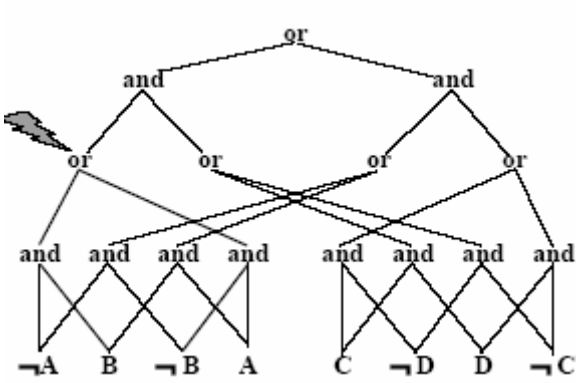


Figure 2.5. Déterminisme.

Exemple 2.10
Soit le nœud '∨' marqué dans la figure 2.5 ayant deux fils tels que chacun d'eux mentionnent les variables A et B : ce nœud est uniforme. Il est de même pour les autres nœuds de disjonction. De ce fait, cette formule est s-NNF.

Il est délicat d'assurer la décomposabilité. Il est aussi délicat d'assurer le déterminisme en préservant la décomposabilité. En revanche, toute formule NNF peut être uniformisée en temps polynomial en préservant la décomposabilité et le déterminisme. Cependant préserver l'aplatissement pourrait donner une taille colossale à la formule NNF. Donc, l'uniformité n'est pas importante d'un point de vue complexité à moins que l'on considère l'aplatissement.

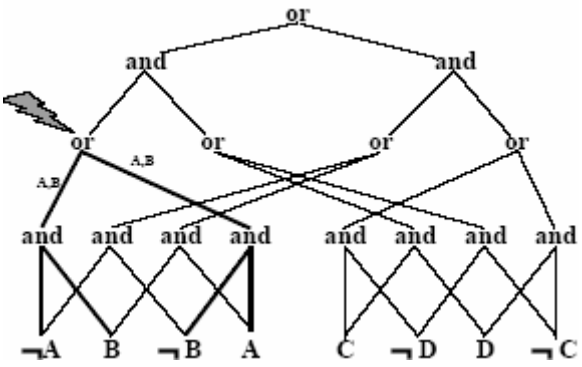


Figure 2.6. Uniformité.

La propriété de décision suivante est issue des diagrammes de décision binaire :

Définition 2.23 (Décision)
Un nœud de **décision** *N* dans une formule NNF est un nœud étiqueté par ⊤, ⊥ ou un nœud de disjonction la forme $(X \wedge \alpha) \vee (\neg X \wedge \beta)$ où X est une variable propositionnelle et α et β sont des nœuds de décision. dVar(N) dénote la variable X. [Bryant 1986]

Définition 2.24 (BDD)
Le langage **BDD** (Binary Decision Diagrams) est le sous-ensemble de NNF tel que la racine de toute formule soit un nœud de décision.

Exemple 2.11
La figure 2.7 représente une formule BDD.

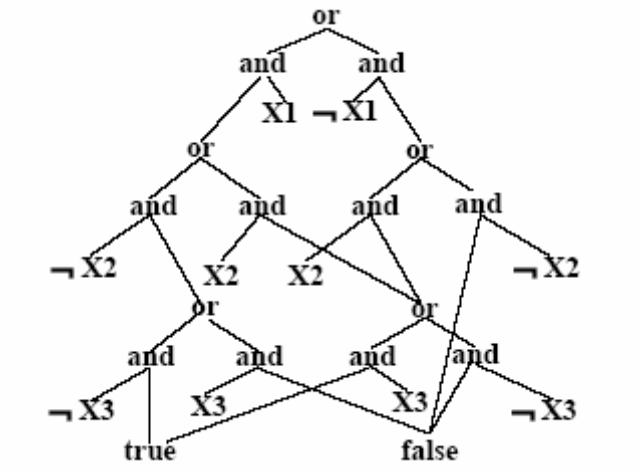


Figure 2.7. Formule BDD.

Le langage BDD correspond aux diagrammes de décision binaire (BDDs pour Binary Decision Diagrams) issus de la littérature de la vérification formelle. Ces derniers sont représentés en utilisant une notation plus compacte où ⊤ et ⊥ sont dénotés par 1 et 0 respectivement et chaque nœud de décision

 est noté par  .

Exemple 2.12
La formule BDD de la figure 2.7 correspond au diagramme de décision binaire de la figure 2.8.

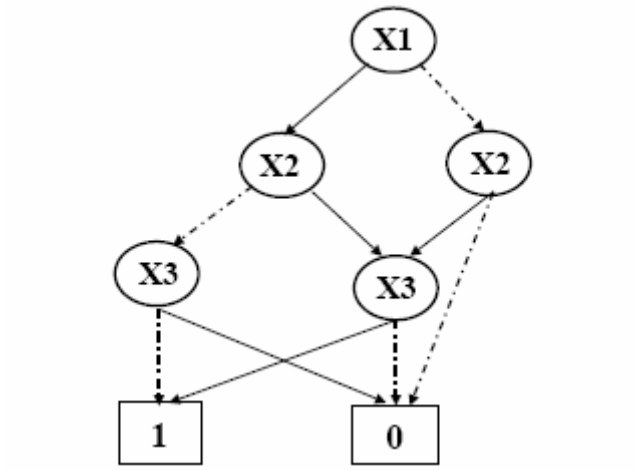


Figure 2.8. Diagramme de décision binaire

Il est clair que toute formule NNF satisfaisant la propriété de décision est une formule déterministe. Donc BDD est un sous ensemble de d-NNF. D'autre part, BDD n'est pas qualifié de langage cible de compilation (à moins que P=NP), mais ses sous ensembles suivants le sont.

Définition 2.25 (FBDD)
Le langage **FBDD** est donné par l'intersection des langages DNNF et BDD.

Le langage FBDD correspond aux diagrammes de décision binaire libre (FBDD pour Free Binary Decision Diagrams) issus aussi de la vérification formelle [Gergov and Meinel 1994]. Un FBDD est souvent défini comme étant un BDD vérifiant la propriété '*lire une seule fois*' (read-once property) : sur chaque chemin entre la racine et une feuille, une variable apparaît au plus une fois. Un sous ensemble du langage FBDD est obtenu en imposant la propriété d'ordre suivante :

Définition 2.26 (OBDD$_<$)
Soit < une relation d'ordre total sur l'ensemble des variables propositionnelles considérées. Le langage **OBDD$_<$** est le sous ensemble de FBDD vérifiant la propriété suivante : si *N* et *M* sont des nœuds de disjonction, et si N est un prédécesseur de M alors dVar(N) < dVar(M).

Définition 2.27 (OBDD)
Le langage **OBDD** est l'union de tous les langages OBDD<. Ce langage correspond aux fameux diagrammes de décision binaire ordonnés (OBDDs pour Ordered Binary Decision Diagrams) [Bryant 1986].

Autrement dit, le langage OBDD est le sous ensemble du langage FBDD où toute formule satisfait la propriété d'ordre : les atomes apparaissent dans le même ordre quelque soit le chemin considéré entre la racine et une feuille. Etant donné un ordre particulier <, le langage OBDD< est le sous ensemble du langage OBDD où toutes les formules respectent le même ordre <. Enfin, on a le langage MODS défini comme suit :

Définition 2.28
MODS (pour Models) est le sous-ensemble de DNF satisfaisant les deux propriétés de déterminisme et d'uniformité.

Exemple 2.13
La figure 2.8 montre une formule MODS.

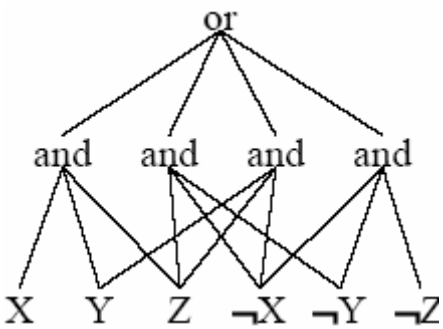


Figure 2.8. Formule MODS.

On fait remarquer qu'à partir de la syntaxe d'une formule MODS, on peut immédiatement déduire ses modèles.

Tous les langages qu'on vient de citer sont représentés sur la figure 2.9. On montre en plus a relation d'inclusion ensembliste qui existe entre eux : une flèche $L_1 \rightarrow L_2$ signifie que L_1 est un sous-ensemble de L_2.

On vient de présenter plus d'une quinzaine de langages dérivés du langage NNF. Alors que certains ont été largement étudiés en informatique, d'autres tels que les langages DNNF et d-DNNF sont relativement récents. La question qui se pose maintenant est la suivante: quel langage choisir pour une application donnée ? En réalité, ce choix est régi par trois critères à savoir la concision du langage choisi, la classe des requêtes auxquelles il permet de répondre ainsi que les transformations qu'il permet d'effectuer et ce en temps polynomial. Dans ce qui suit, on définit précisément et formellement ces critères et surtout on analyse les langages précédents en fonction d'eux.

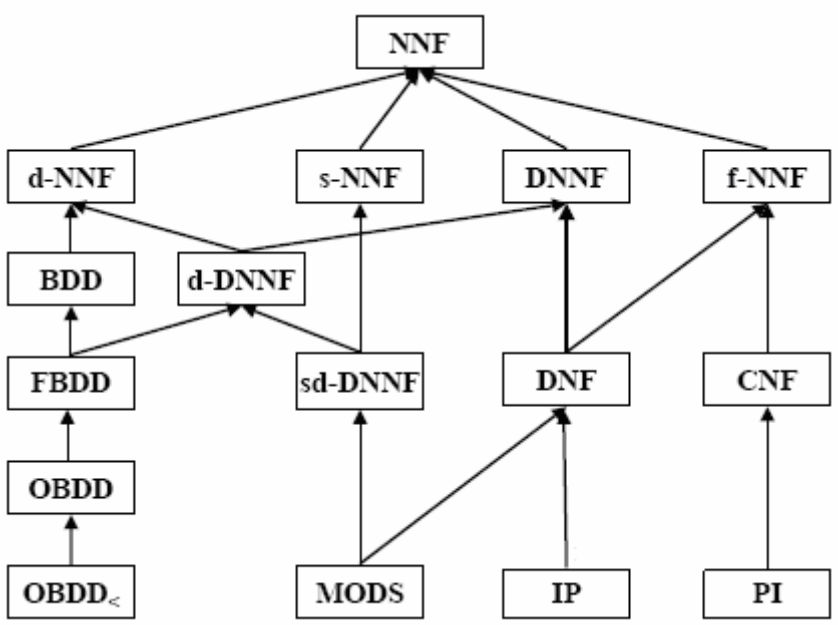


Figure 2.9. Langages dérivés de NNF.

2.4.2. Concision des langages cibles de compilation

Définition 2.29 (Concision)
Soient L_1 et L_2 deux sous ensembles du langage NNF. L_1 est dit **au moins aussi concis que** L_2, noté $L_1 \leq L_2$, si et seulement si pour toute formule $\alpha \in L_2$, il existe une formule $\beta \in L_1$ qui lui est équivalente, de plus la taille de β est polynomiale par rapport à la taille de α. [Gogic *et al*. 1995]

On remarque que la relation $\leq$ est réflexive et transitive. D'autre part on peut définir la relation $<$ telle que $L_1 < L_2$ si et seulement si $L_1 \leq L_2$ et L_2 n'est pas $\leq L_1$.

Proposition 2.1
La relation de concision entre les langages dérivés du langage NNF est exhibée par la table 2.1.

On note que l'occurrence de '$\leq$' dans la case (i, j) signifie que $L(i, 1) \leq L(1, j)$ et l'occurrence de '*' se lit à moins que P = NP. De plus, l'occurrence de ' ?' reflète l'ignorance.

Un résultat classique en compilation de connaissances stipule qu'il n'est pas possible de compiler n'importe quelle formule propositionnelle α en une structure de données de taille polynomiale β tel que α et β infèrent le même ensemble de clauses, et la déduction clausale dans β se fait en temps polynomial par rapport à sa taille à moins que P = NP. Ce résultat est utilisé dans certaines preuves de cette proposition.

L	NNF	DNNF	d-DNNF	sd-DNNF	FBDD	OBDD	OBDD<	DNF	CNF	PI	IP	MODS
NNF	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$
DNNF	$>$*	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\not\leq$*	?	$\leq$	$\leq$
d-DNNF	$\not\leq$*	$\not\leq$*	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\not\leq$*	$\not\leq$*	?	?	$\leq$
sd-DNNF	$\not\leq$*	$\not\leq$*	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\not\leq$*	$\not\leq$*	?	?	$\leq$
FBDD	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\leq$	$\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$
OBDD	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$
OBDD<	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$
DNF	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\not\leq$	$\not\leq$	$\leq$	$\leq$
CNF	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\leq$	$\not\leq$	$\leq$
PI	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\not\leq$	?
IP	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$	$\leq$
MODS	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\not\leq$	$\leq$

Table 2.1. Concision des langages dérivés de NNF.

La figure 2.10 résume graphiquement la proposition 2.1. Une flèche $L_1 \rightarrow L_2$ indique que $L_1 < L_2$. Une arête discontinue indique une relation inconnue. Excepté le langage PI, le langage DNNF s'avère le plus concis par rapport aux autres langages cibles de compilation. En fait, on sait que PI n'est pas plus concis que DNNF mais on ignore l'inverse. Entre les langages DNNF et MODS, on a la relation de concision suivante : DNNF < d-DNNF < FBDD < OBDD < $\text{OBDD}_<$ < MODS.

DNNF est obtenu en imposant la propriété de décomposabilité sur le langage NNF; d-DNNF en ajoutant le déterminisme; FBDD en ajoutant la décision et OBDD et $\text{OBDD}_<$ en ajoutant l'ordre (n'importe quel ordre dans le premier cas et un ordre particulier dans le second). Ajouter chacune de ces propriétés réduit la concision du langage en question. En revanche, ajouter la régularité au langage d-DNNF ne change pas sa concision : les deux langages d-DNNF et sd-DNNF sont au même pieds d'égalité quant à leur concision.

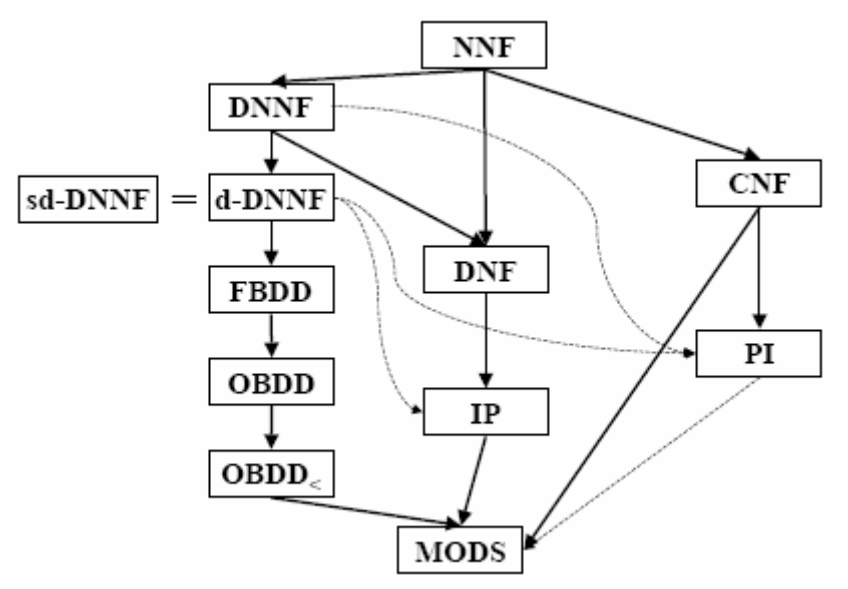


Figure 2.10. Concision des langages cibles de compilation de connaissances.

2.4.3. Requêtes logiques et langages dérivés de NNF

On définit dans un premier temps un ensemble de requêtes logiques, chacune d'entre elles retourne une information à propos de la base de connaissances en question. Ensuite on identifie les langages cibles de compilations qui permettent de répondre à ces requêtes en temps polynomial. Les requêtes qui sont considérées sont le test de consistance, de validité, d'impliqués (déduction clausale), d'impliquants, d'équivalence, et de déduction de formules. On considère aussi le comptage et l'énumération de modèles.

Soit *L* un sous ensemble du langage NNF.

Définition 2.30 (CO)
L satisfait **CO** si et seulement s'il existe un algorithme polynomial permettant de déterminer pour toute formule Σ de *L* si elle est consistante ou non.

Définition 2.31 (VA)
L satisfait **VA** si et seulement s'il existe un algorithme polynomial permettant de déterminer pour toute formule Σ de *L* si elle est valide ou non.

Une des motivations majeures de la compilation de connaissances est de permettre de répondre plus efficacement à la déduction clausale.

Définition 2.32 (CE)
L satisfait **CE** si et seulement s'il existe un algorithme polynomial permettant de déterminer pour toute formule Σ de *L* et pour toute clause *c* si Σ |= *c* ou non.

Définition 2.33 (EQ)
L satisfait **EQ** si et seulement s'il existe un algorithme polynomial permettant de déterminer pour toutes formules Σ et Φ de *L* si $\Sigma \equiv \Phi$ ou non.

Définition 2.34 (SE)
L satisfait **SE** si et seulement s'il existe un algorithme polynomial permettant de déterminer pour toutes formules Σ et Φ de *L* si $\Sigma \models \Phi$ ou non.

On note que la déduction de formules est plus forte que la déduction clausale et l'équivalence. Donc, si un langage *L* satisfait SE, il satisfait aussi CE et EQ.
Maintenant, on considère la requête duale de CE:
Définition 2.35 (IM)
L satisfait **IM** si et seulement s'il existe un algorithme polynomial qui permet de déterminer pour toute formule Σ de *L* et pour tout terme *t* si $t \models \Sigma$ ou non.

Enfin, on considère le comptage et l'énumération de modèles :

Définition 2.36 (CT)
L satisfait **CT** si et seulement s'il existe un algorithme polynomial qui permet de déterminer pour toute formule Σ de *L* le nombre de ses modèles.

Définition 2.37 (ME)
L satisfait **ME** si et seulement s'il existe un polynôme p(., .) et un algorithme qui donne pour toute formule Σ de *L* tous ses modèles en temps p(*n*,*m*), où *n* est la taille de la formule Σ et *m* le nombre de ses modèles.

Proposition 2.2
La relation de satisfaction d'un langage pour une requête est donnée par la table 2.2.

Chacun des langages NNF, s-NNF, d-NNF, f-NNF et BDD appartiennent à la même classe qui ne permet de répondre à aucune requête en temps polynomial, CNF satisfait uniquement VA et IM si bien qu'aucun de ces langages n'est qualifié de langage cible de compilation. Sinon, tous les autres langages satisfont CO et CE.

On rappelle l'ordre de concision suivant : DNNF < d-DNNF < FBDD < OBDD < $OBDD_<$. DNNF satisfait immédiatement CO, CE et ME. En ajoutant le déterminisme à la décomposabilité (d-DNNF), on satisfait de plus VA, IM et CT qui sont assez importants. Cependant, on ignore toujours si d-DNNF satisfait EQ ou non. D'autre part, ajouter la décision à la décomposabilité et au déterminisme (FBDD), n'augmente pas l'efficacité, par contre, ça réduit la concision. Enfin, ajouter la propriété d'ordre à la décomposabilité, au déterminisme et à la décision permet de satisfaire SE (pour OBDD et $OBDD_<$) et EQ (pour $OBDD_<$).

L	CO	VA	CE	IM	EQ	SE	CT	ME
NNF	○	○	○	○	○	○	○	○
DNNF	✓	○	✓	○	○	○	○	✓
d-NNF	○	○	○	○	○	○	○	○
s-NNF	○	○	○	○	○	○	○	○
f-NNF	○	○	○	○	○	○	○	○
d-DNNF	✓	✓	✓	✓	?	○	✓	✓
sd-DNNF	✓	✓	✓	✓	?	○	✓	✓
BDD	○	○	○	○	○	○	○	○
FBDD	✓	✓	✓	✓	?	○	✓	✓
OBDD	✓	✓	✓	✓	✓	○	✓	✓
OBDD<	✓	✓	✓	✓	✓	✓	✓	✓
DNF	✓	○	✓	○	○	○	○	✓
CNF	○	○	○	✓	○	○	○	○
PI	✓	✓	✓	✓	✓	✓	○	✓
IP	✓	✓	✓	✓	✓	✓	○	✓
MODS	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.2. Langages dérivés de NNF et requêtes logiques. '○' signifie 'ne satisfait pas à moins que NP=P.

Quant à l'ordre de concision NNF < DNNF < DNF < IP < MODS, il est à noter que DNNF est obtenu en imposant la propriété de décomposabilité, tandis que DNF est obtenu en imposant les propriétés d'aplatissement et la conjonction simple (qui est plus forte que la décomposabilité). Ce qui est intéressant est que DNNF est aussi moins concis que DNF tout en supportant le même ensemble de requêtes en temps polynomial. D'autre part, l'ajout de l'uniformité et du déterminisme au DNF ce qui donne MODS assure cinq requêtes supplémentaires entre autres SE et EQ.

Enfin, on note que le déterminisme est nécessaire (mais pas suffisant) pour CT : seuls les langages déterministes (d-DNNF, sd-DNNF, FBDD, $OBDD_<$, OBDD et MODS) satisfont CT. De plus, CT implique VA mais l'inverse n'est pas vrai.

2.4.4. Transformations logiques et langages dérivés de NNF

Une requête est une opération qui retourne une information à propos de la base de connaissances sans pour autant la modifier. Une transformation, par contre, est une opération qui retourne une base modifiée. De nombreuses applications requièrent une combinaison de transformations et de requêtes. On commence par donner les définitions des transformations suivantes :

Définition 2.38 ($\wedge$C)
L satisfait $\wedge$**C** si et seulement s'il existe un algorithme polynomial qui associe à tout ensemble fini de formules $\Sigma_1,\ldots, \Sigma_n$ de *L* une formule de *L* qui soit logiquement équivalente à $\Sigma_1 \wedge \ldots \wedge \Sigma_n$.

Définition 2.39 ($\vee$C)
L satisfait $\vee$**C** si et seulement s'il existe un algorithme polynomial qui associe à tout ensemble fini de formules $\Sigma_1,\ldots, \Sigma_n$ de *L* une formule de *L* qui soit logiquement équivalente à $\Sigma_1 \vee \ldots \vee \Sigma_n$.

Définition 2.40 ($\neg$C)
L satisfait **($\neg$C)** si et seulement s'il existe un algorithme polynomial qui associe à toute formule Σ de *L* une formule de *L* équivalente à $\neg\Sigma$.

Un langage satisfaisant une des transformations précédentes est dit *clos* par rapport à cette transformation. La fermeture par rapport aux transformations logiques est importante pour deux raisons. D'une part, ça influence la façon avec laquelle on construit un compilateur d'un langage cible donné. Par exemple, si une clause est facilement compilable vers un langage *L*, alors la fermeture par rapport à la conjonction implique que la compilation d'une formule CNF vers *L* soit facile aussi. D'autre part, cela a des conséquences sur l'ensemble des requêtes que supporte le langage cible en temps polynomial : si un langage *L* satisfait CO et est clos par rapport à la négation et par rapport à la conjonction, alors il doit satisfaire aussi SE (pour vérifier si $\Sigma \models \Phi$, par le biais du théorème de réfutation, ça revient à vérifier si $\Phi \wedge \neg \Sigma$ est inconsistant). Par analogie, si un langage *L* satisfait VA et est clos par rapport à la négation et par rapport à la disjonction, alors il doit satisfaire aussi SE (pour vérifier si $\Sigma \models \Phi$, par le biais du théorème de déduction, ça revient à vérifier si $\neg \Sigma \vee \Phi$ est valide).

Il importe de préciser que certains langages sont clos par rapport à un opérateur logique si seulement le nombre d'opérandes est borné par une constante et on parle dans ce cas de *fermeture bornée.*

Définition 2.41 ($\wedge$BC)
L satisfait $\wedge$**BC** si et seulement s'il existe un algorithme polynomial qui associe à toute paire de formules Σ et Φ de *L* une formule de *L* équivalente à $\Sigma \wedge \Phi$.

Définition 2.42 ($\vee$BC)
L satisfait $\vee$**BC** si et seulement s'il existe un algorithme polynomial qui associe à toute paire de formules Σ et Φ de *L* une formule de *L* équivalente à $\Sigma \vee \Phi$.

Le conditionnement est une autre transformation intéressante qu'on définit par :

Définition 2.43 (Conditionnement)
Soit Σ une formule propositionnelle, soit *t* un terme consistant. Le **conditionnement** de Σ sur *t*, noté par $\Sigma \mid t$, est la formule obtenue en remplaçant toute occurrence d'une variable x dans Σ par VRAI (resp. FAUX) si le littéral correspondant apparaît positif dans *t* (resp. négatif).

Définition 2.44 (CD)
L satisfait **CD** si et seulement s'il existe un algorithme polynomial qui associe à toute formule Σ de *L* et à tout terme consistant *t* une formule de *L* équivalente à $\Sigma \mid t$.

Le conditionnement a de nombreuses applications, et il correspond à la *restriction* dans la littérature des fonctions booléennes. L'application principale du conditionnement est inhérente à un

théorème qui stipule que $\Sigma \wedge t$ est consistant si et seulement $\Sigma \mid t$ est consistant [Darwiche 2001a]. Donc si un langage satisfait CO et CD alors il satisfait aussi CE. Le conditionnement joue aussi un rôle important dans la construction de compilateurs DNNF. En fait, si deux formules Σ_1 et Σ_2 sont toutes les deux décomposables, leur conjonction $\Sigma_1 \wedge \Sigma_2$ n'est pas nécessairement décomposable étant donné que ces formules peuvent avoir des variables communes. Le conditionnement peut alors être utilisé dans ce cas du fait que $\Sigma_1 \wedge \Sigma_2$ est équivalent à $\vee_t (\Sigma_1 \mid t) \wedge (\Sigma_2 \mid t) \wedge t$, où t est un terme construit à partir de toutes les variables communes entre Σ_1 et Σ_2. Cette proposition est la base de la compilation CNF vers DNNF qu'on verra en détail dans la section suivante.

Enfin, on définit la transformation *d'oubli de variables* :

Définition 2.45 (Oubli de variables)
Soit Σ une formule propositionnelle, et soit A un ensemble fini de variables propositionnelle. L'oubli de A dans Σ, dénoté par $\exists A. \Sigma$ (ou ForgetVariable(Σ , A)) est une formule qui ne contient aucune variable de A et telle que pour toute formule Ψ qui ne mentionne aucune variable de A ($Var(\Psi) \cap A = \varnothing$), on a $\Sigma \models \Psi$ si et seulement si $\exists A. \Sigma \models \Psi$. [Lin and Reiter 1994]

Oublier les variables de A dans Σ consiste à éliminer toute référence de A dans Σ en maintenant toute information que fournit Σ à propos du complément de A.

Définition 2.46 (FO, SFO)
L satisfait **FO** si et seulement s'il existe un algorithme polynomial qui associe à toute formule Σ de L et à tout ensemble A de variables une formule de L équivalente à $\exists A. \Sigma$. Si on considère un singleton A, on dit que L satisfait **SFO**.

L'oubli est une transformation importante étant donné qu'elle permet de projeter une base de connaissances sur un ensemble de variables. Par exemple, si on sait d'emblée que certaines variables A ne vont pas apparaître dans les requêtes, on peut alors oublier ces variables dans la base compilée tout en préservant sa capacité de répondre correctement à ces requêtes. En outre, ça a de nombreuses applications en planification et en diagnostic. Cette transformation sera traitée plus en détail dans le chapitre 3.

Proposition 2.3
Soit L un langage dérivé de NNF. Les transformations logiques par rapport auxquelles il est clos sont présentées sur la table 2.3.

On constate que tous les langages satisfont CD. Quant à FO, seuls DNNF, DNF, PI et MODS le satisfont. Il est à noter qu'aucun des langages FBDD, OBDD et $OBDD_<$ ne satisfait FO. Néanmoins, certaines applications récentes du langages $OBDD_<$ en planification dépendent essentiellement de l'oubli donc il est plus judicieux d'utiliser un langage qui satisfait FO dans ce cas là. En revanche, OBDD et OBDD< satisfont SFO alors que FBDD ne le fait pas.

On constate aussi qu'aucun des langages cibles de compilation n'est clos par rapport à la conjonction. Par contre, certains d'entre eux sont clos par rapport à la conjonction bornée à savoir les langages $OBDD_<$, DNF, IP et MODS. Quant à la disjonction, les seuls langages cibles de compilation

qui sont clos par rapport à elle sont DNNF et DNF. Les langages OBDD et PI sont par contre clos par rapport à la disjonction bornée. Les seuls langages cibles de compilation clos par rapport à la négation sont FBDD, OBDD et $OBDD_<$. On note aussi que les langages d-DNNF et FBDD supportent le même ensemble de requêtes en temps polynomial, donc ils ne sont pas différents de ce point de vue. La seule différence entre eux dans la table 2.3 est la fermeture du langage FBDD par rapport à la négation qui ne semble pas très importante étant donnée sa non fermeture par rapport à la conjonction ou à la disjonction. De plus, d-DNNF est plus concis que FBDD.

L	CD	FO	SFO	∧C	∧BC	∨C	∨BC	¬C
NNF	✓	o	✓	✓	✓	✓	✓	✓
DNNF	✓	✓	✓	o	o	✓	✓	o
d-NNF	✓	o	✓	✓	✓	✓	✓	✓
s-NNF	✓	o	✓	✓	✓	✓	✓	✓
f-NNF	✓	o	✓	•	•	•	•	✓
d-DNNF	✓	o	o	o	o	o	o	?
sd-DNNF	✓	o	o	o	o	o	o	?
BDD	✓	o	✓	✓	✓	✓	✓	✓
FBDD	✓	•	o	•	o	•	o	✓
OBDD	✓	•	✓	•	o	•	o	✓
OBDD<	✓	•	✓	•	✓	•	✓	✓
DNF	✓	✓	✓	•	✓	✓	✓	•
CNF	✓	o	✓	✓	✓	•	✓	•
PI	✓	✓	✓	•	•	•	✓	•
IP	✓	•	•	•	✓	•	•	•
MODS	✓	✓	✓	•	✓	•	•	•

Table 2.3. Langages dérivés de NNF et transformations logiques. • signifie 'ne satisfait pas' tandis que o signifie ne satisfait pas à moins que P = NP.

Enfin, $OBDD_<$ est l'unique langage cible de compilation clos par rapport à la négation, la conjonction bornée et la disjonction bornée. Cette fermeture joue un rôle important dans la compilation des bases de connaissances propositionnelles vers le langage OBDD< et est la base de l'état de l'art des compilateurs dédiés à cette fin [Bryant 1986].

2.5. Compilateur DNNF

Le langage DNNF (pour Decomposable Negation Normal Form) est un langage cible de compilation de connaissances relativement récent. Sa première investigation était dans le contexte du diagnostic basé modèles [Darwiche 1998]. Ensuite, on a constaté que son utilité est indépendante de cette application spécifique. En effet, il peut être vu comme un langage cible de compilation intéressant. L'ensemble d'opérations (requêtes et transformations) logiques qu'il supporte en temps polynomial (CO, CE, ME, CD, FO, SFO, ∨C, ∨BC) est relativement suffisant pour réaliser de nombreuses applications complexes en Intelligence Artificielle par exemple en diagnostic et en

planification. En outre, hormis le langage PI, le langage DNNF est le plus concis de tous les langages cibles de compilation de connaissances (on sait que PI n'est pas plus concis que DNNF mais on ignore l'inverse).

On rappelle que le langage DNF est inclus dans le langage DNNF (toute formule DNF est une formule DNNF), donc toutes les propriétés du langage DNNF sont aussi valables pour le langage DNF. Une question pertinente se pose alors: pourquoi ne pas compiler une base de connaissances vers le langage DNF au lieu de la compiler vers le langage DNNF ? En réalité, c'est dû au fait qu'il existe des bases qui ont une représentation polynomiale en DNNF, mais qui est exponentielle en DNF. Ce n'est n'est rien d'autre que la traduction du fait que le langage DNNF soit plus concis que le langage DNF.

Ces caractéristiques intéressantes du langage DNNF ont motivé Darwiche à concevoir un compilateur d'une base de connaissances sous forme CNF vers une autre base sous forme DNNF. Le compilateur proposé dans [Darwiche 2001a] s'articule autour de la proposition suivante :

Proposition 2.4
Soient Δ_1 et Δ_2 deux formules DNNF et soit $X = \text{Var}(\Delta_1) \cap \text{Var}(\Delta_2)$.
Soit Δ la formule donnée par $\vee_\beta (\Delta_1 \mid \beta) \wedge (\Delta_2 \mid \beta) \wedge \beta$, où β est un terme construit à partir des variables de X. On montre alors que Δ est une formule DNNF équivalente à $\Delta_1 \wedge \Delta_2$.

On illustre cette proposition via l'exemple suivant :

Exemple 2.14
Soient $\Delta = (\neg A \vee B) \wedge (\neg B \vee C)$.
Selon la proposition 2.4, la formule :
$[((\neg A \vee B) \mid B) \wedge ((\neg B \vee C) \mid B) \wedge B] \vee [((\neg A \vee B) \mid \neg B) \wedge ((\neg B \vee C) \mid \neg B) \wedge \neg B] \equiv (C \wedge B) \vee (\neg A \wedge \neg B)$ est une formule DNNF qui est équivalente à $\mathbf{\Delta}$.

Etant donnée une formule CNF Δ, l'idée du compilateur est de considérer Δ comme une conjonction de deux sous-formules $\mathbf{\Delta}_1$ et $\mathbf{\Delta}_2$ ensuite appliquer la proposition 2.4 ce qui donne $\vee_\beta (\Delta_1 \mid \beta) \wedge (\Delta_2 \mid \beta) \wedge \beta$. Pour chaque β, si $(\Delta_1 \mid \beta)$ est une clause alors on ne fait rien, sinon on refait le même traitement qu'on a effectué pour Δ et ainsi de suite. Il est de même pour $(\Delta_2 \mid \beta)$. Néanmoins, cette procédure récursive n'est pas déterministe : comment choisir Δ_1 et Δ_2 ? La solution réside dans l'utilisation d'un *arbre de décomposition*.

Définition 2.47 (Arbre de décomposition)
Un **arbre de décomposition** T d'une forme CNF $\mathbf{\Delta}$ est un arbre binaire dont les feuilles correspondent aux clauses de $\mathbf{\Delta}$. Si t est la feuille de T qui correspond à une clause α dans $\mathbf{\Delta}$, alors $\mathbf{\Delta}(t) = \{\alpha\}$.

Pour chaque nœud interne t :

- t_l dénote son fils gauche et t_r son fils droit ;
- $\mathbf{\Delta}(t) = \mathbf{\Delta}(t_l) \cup \mathbf{\Delta}(t_r)$;
- $\text{Var}(t) = \text{Var}(\mathbf{\Delta}(\text{t}))$;

- $\text{Var}^{\uparrow}(t)$ est défini par l'ensemble des variables associées aux feuilles qui ne sont pas dans le sous arbre de racine t.

Exemple 2.15
La figure 2.11 décrit un arbre de décomposition de la formule CNF $\mathbf{\Delta} = \{\neg A \vee B, \neg B \vee C, \neg C \vee D\}$.
$\mathbf{\Delta}(t_2) = \{\neg A \vee B, \neg B \vee C\}$
$\text{Var}(t_2) = \{A, B, C\}$
$\text{Var}^{\uparrow}(t_2) = \{C,D\}$ et $\text{Var}^{\uparrow}(t_4) = \{A, B, C, D\}$.

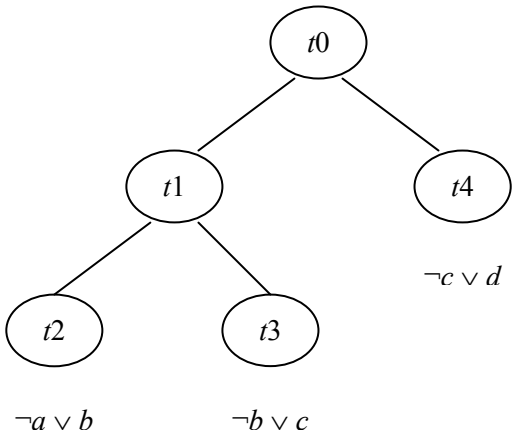


Figure 2.11. Arbre de décomposition.

Soit un arbre de décomposition T (de racine t_0) d'une formule CNF $\mathbf{\Delta}$, la compilation de $\mathbf{\Delta}$ vers le langage DNNF se fait par l'appel dnnf1(t_0, VRAI) où dnnf1 est la procédure définie comme suit :

Procédure 2.1 dnnf1(t, a)
/* t est un nœud dans un arbre de décomposition */
/* a est un terme */
début

si t est une feuille et $\Delta(t) = \{c\}$ **alors** $\delta \leftarrow c \mid a$;
sinon $\delta \leftarrow \vee_{\beta}$ dnnf1(tl, $a \wedge \beta$) $\wedge$ dnnf1(tl, $a \wedge \beta$) $\wedge \beta$;
/* β est une instanciation construite sur $\text{Var}(t_l) \cap \text{Var}(t_r) - \text{Var}(a)$ */
retourner δ ;

fin

D'autre part, on montre la proposition suivante :

Proposition 2.5
Soit t un nœud et soient α et α' deux termes. Si $\text{Var}(\alpha) \cap \text{Var}(t) = \text{Var}(\alpha') \cap \text{Var}(t)$ alors dnnf1(t,α) est équivalent à dnnf1(t,α').

Par conséquent, il est intéressant d'associer à chaque nœud t un cache qui mémorise le résultat de dnnf1(t, α) en l'indexant par le sous terme de α contenant des variables de Var(t). Ainsi, lorsque l'on a un appel récursif dnnf1(t, α), on consulte d'abord le cache du nœud t pour vérifier si une entrée pour sous_terme(α, Var(t)) existe et dans ce cas on ne refait pas le calcul. En tenant compte de cette amélioration, le fonctionnement du compilateur DNNF est décrit par la procédure suivante :

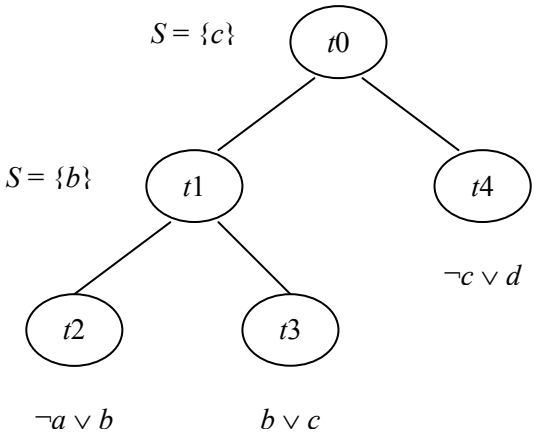
Procédure 2.2 dnnf(*t, a*)
début

/* t est un nœud dans un arbre de décomposition */
/* a est un terme */
/* $cache_t(.)$ est initialisée à nil */

$\Psi \leftarrow$ sous_terme(α, Var(t)) ;
si $cache_t(\Psi) \neq$ nil **alors retourner** $cache_t(\Psi)$;
si t est une feuille et $\Delta(t) = \{c\}$ **alors** $\delta \leftarrow c \mid a$;
sinon $\delta \leftarrow \vee_\beta$ dnnf1(tl, $a \wedge \beta$) $\wedge$ dnnf1(tl, $a \wedge \beta$) $\wedge \beta$;
/* β est une instanciation construite sur $Var(t_l) \cap Var(t_r) - Var(a)$ */
$cache_t(\Psi) \leftarrow \delta$;
retourner δ ;

fin

Exemple 2.16
Soit $\Delta = (\neg a \vee b) \wedge (b \vee c) \wedge (\neg c \vee d)$.
Soit l'arbre de décomposition suivant :



Afin de compiler Δ vers le langage DNNF, on appelle la procédure dnnf(t0, VRAI), on obtient :
dnnf(t0, VRAI) = [$c \wedge$ dnnf(t1, c) $\wedge$ dnnf(t4, c)] $\vee$ [$\neg c \wedge$ dnnf(t1, $\neg c$) $\wedge$ dnnf(t4, $\neg c$)]
= [$c \wedge$ d $\wedge$ dnnf(t1, c)] $\vee$ [$\neg c \wedge$ dnnf(t1, $\neg c$)]

On calcule alors dnnf(t1, c) :

dnnf(t1, c) = [$b \wedge$ dnnf(t2, $c \wedge b$) $\wedge$ dnnf(t3, $c \wedge b$)] $\vee$ [$\neg b \wedge$ dnnf(t2, $c \wedge \neg b$) $\wedge$ dnnf(t3, $c \wedge \neg b$)]

= b $\vee$ ($\neg b \wedge \neg a$)

On calcule aussi dnnf(t1, $\neg c$)

dnnf(t1, $\neg c$) = [$b \wedge$ dnnf(t2, $\neg c \wedge b$) $\wedge$ dnnf(t3, $\neg c \wedge b$)] $\vee$ [$\neg b \wedge$ dnnf(t2, $\neg c \wedge \neg b$) $\wedge$ dnnf(t3, $\neg c \wedge \neg b$)]

= b

Enfin dnnf(t0, VRAI) = ($c \wedge d \wedge (b \vee (\neg b \wedge \neg a))$) $\vee$ ($\neg c \wedge b$) qui n'est rien d'autre que la compilation DNNF de Δ.

D'un autre côté, on constate que l'analyse des cas effectuée sur l'ensemble les variables communes entre les sous-formules Δ_1 et Δ_2 peut augmenter considérablement la taille de la formule compilée. Avant de donner la complexité de l'algorithme dnnf, on donne les définitions suivantes :

Définition 2.48 (Cluster)

Soit t un nœud dans un arbre de décomposition T. Le **cluster** du nœud t est défini comme suit :

- Si t est une feuille, alors cluster(t) = Var(t).
- Si t est un nœud interne, alors cluster(t) = (Var(t) $\cap$ Var$^{\uparrow}$(t)) $\cup$ (Var(t_l) $\cap$ Var(t_r)).

Définition 2.49 (Largeur)

La **largeur** d'un arbre de décomposition est la taille de son maximum cluster moins 1.

Exemple 2.17

Soit l'arbre de décomposition de l'exemple 2.15.

cluster(t_1) = {C}, cluster(t_2) = {B, C}, cluster(t_3) = {A, B}, cluster(t_4) = {B, C}, cluster(t_5) = {C, D}.

La largeur de cet arbre de décomposition est de 1.

On donne maintenant la complexité de la procédure dnnf :

Proposition 2.6

Soit Δ une formule CNF ayant n clauses. Soit T un arbre de décomposition de Δ de largeur w. L'algorithme dnnf requiert $O(nw2^w)$ temps et espace.

D'après cette proposition, la largeur de l'arbre de décomposition utilisé est un facteur critique pour l'efficacité du compilateur. La construction de 'bons' arbres de décompositions c'est-à-dire d'une largeur qui n'est pas importante fait l'objet d'autres travaux. Dans [Darwiche and Hopkin 2001], on présente une méthode de constructions d'arbres de décomposition basée sur le la décomposition d'un hypergraphe. Une autre méthode, qu'on va décrire succinctement, s'appuie sur un ordre d'élimination de variables. Etant donné un ordre π sur les variables de la formule **Δ,** on commence par créer un arbre binaire contenant un seul nœud t pour chaque clause α dans **Δ**. Ensuite, et selon l'ordre π, pour toute variable X de **Δ,** on combine (arbitrairement) tous les arbres binaires qui contiennent X en un seul arbre binaire. Une fois on a considéré toutes les variables, on combine encore tous les arbres générés en un seul arbre binaire qui représente l'arbre de décomposition. La procédure 2.3 décrit formellement cette méthode :

Procédure 2.3 dtree(Δ, π)
/* Δ une forme CNF */
/* π un ordre sur les variables de Δ */
/* compose (Γ) combine arbitrairement les arbres de Γ en un seul arbre binaire */
/* compose($\{T\}$) = T */
/* compose($\{T_1, T_2\}$) est un arbre binaire ayant T_1 et T_2 comme fils */
/* compose ($\{T_1, \dots, T_n\}$) = compose ($\{T_1$, compose($\{T_2, \dots, T_n\}$)) tel que n > 2 */
début

$\Sigma \leftarrow \{T_a : a \in \Delta\}$ tel que T_a est un arbre de décomposition contenant un nœud unique t et $\Delta(t) = \{a\}$
pour i ← 1 **à** taille de π **faire**
    $\Gamma \leftarrow \{T : T \in \Sigma$, Var $\pi(i)$ apparaît dans l'arbre $T\}$;
    $\Sigma \leftarrow (\Sigma \setminus \Gamma) \cup \{$compose($\Gamma$)$\}$;
fait
retourner compose(Σ) ;
fin

Afin de borner la largeur de l'arbre de décomposition généré en utilisant cette méthode, il convient de définir au préalable ce qu'un *graphe d'interaction* et ce que le *treewidth* d'un ordre d'élimination de variables par rapport à un graphe d'interaction.

Définition 2.50 (Graphe d'interaction)
Etant donnée une formule CNF **Δ**. Le **graphe d'interaction** de **Δ** est le graphe non orienté G construit comme suit : les nœuds de G sont les variables de Δ et il existe une arête entre deux variables (nœuds) de G si et seulement si ces variables apparaissent dans une même clause de **Δ** [Dechter and Rish 1994].

Définition 2.51 (Treewidth)
Soit G un graphe non orienté et soit π un ordre sur ses nœuds. Eliminer un nœud de G revient à connecter tous ses voisins ensuite l'éliminer de G. Si les nœuds sont éliminés de G selon l'ordre π, alors le **treewidth** de π par rapport au graphe G est donné par le nombre maximum de voisins de n'importe quel nœud de G avant qu'il ne soit éliminé [Dechter and Rish 1994].

Enfin, on montre la proposition suivante :

Proposition 2.7
Soient Δ une forme CNF, G son graphe d'interaction, π un ordre de ses nœuds et h le treewidth de π par rapport au graphe G. L'appel de dtree(Δ, π) retourne un arbre de décomposition de Δ de largeur ≤ h.

Le treewidth du meilleur ordre π d'un graphe G (celui qui donne le treewidth le plus petit) est dit le treewidth de G [Bodlaender 1993]. On déduit que si une formule CNF Δ admet un graphe d'interaction d'un treewidth borné, alors calculer un arbre de décomposition optimal pour cette formule et la compiler en se basant sur l'arbre de décomposition généré se font en temps polynomial.

2.6. De SAT à la compilation de connaissances

Le problème de satisfiabilité d'une formule propositionnelle, SAT en abrégé, constitue la pierre angulaire des problèmes NP-complets. Outre que son importance théorique, SAT suscite un intérêt particulier dans divers domaines notamment l'intelligence artificielle, la théorie des graphes et la recherche opérationnelle.

Dans [Huang and Darwiche 2005], on exhibe un lien étroit entre la résolution du problème SAT et la compilation de connaissances en logique propositionnelle. En fait, la plupart des algorithmes complets pour SAT (hormis ceux basés sur la résolution) ne sont que des variantes de la procédure de Davis et Putnam (DP) [Davis *et al*. 1962]. La trace d'une version exhaustive de cette procédure mémorisée d'une façon compacte sur un DAG peut constituer une compilation de la formule en question. En imposant ou en éliminant certaines contraintes, la formule compilée appartiendra à un des langages suivants : d-DNNF, FBDD ou OBDD.

2.6.1. Procédure de Davis & Putnam

On présente en premier lieu la procédure DP dans une version qui se veut la plus simple et la plus élémentaire possible.

Procédure 2.4 DP(Δ)
/* Δ une formule CNF */
début

si il existe une clause vide dans Δ alors **retourner** 0 ;
si il n'y a pas de variable dans Δ alors **retourner** 1 ;
sélectionner une variable x de Δ ;
retourner DP($\Delta_{(x=0)}$) ou DP($\Delta_{(x=1)}$) ;

fin

Dans cette procédure, les variables sont assignées variable par variable. Le choix de la prochaine variable à affecter se fait selon une heuristique de branchement. $\Delta_{(x=0)}$ (resp. $\Delta_{(x=1)}$) désigne la formule Δ simplifiée après avoir remplacé toutes les occurrences de x par VRAI (resp. FAUX). Ensuite on effectue les tests suivants :

- S'il existe une clause vide (ne contenant aucun littéral), alors on retourne 0 (Δ n'est pas satisfiable).
- S'il n'existe plus de variables non affectées, alors on retourne 1 (Δ est satisfiable).
- Sinon continuer l'instanciation des autres variables en réaffectant à la dernière variable instanciée par FAUX la valeur VRAI.

Cette procédure effectue une recherche dans l'espace des assignations et s'arrête lorsqu'elle trouve une assignation qui satisfait la formule CNF ou réalise qu'une telle assignation n'existe pas. La version exhaustive de la procédure DP ne s'arrête pas sur la première instanciation trouvée. En fait,

elle énumère toutes les assignations satisfaisant la formule en explorant dans tous les cas les deux branches à la ligne 6

Exemple 2.18
La figure 2.12 représente la trace de la version exhaustive de la procédure DP appliquée à la formule CNF suivante :

$$\begin{cases} x_1 \vee x_2 \\ x_1 \vee \neg x_2 \vee x_3 \\ \neg x_1 \vee x_2 \vee x_3 \end{cases}$$

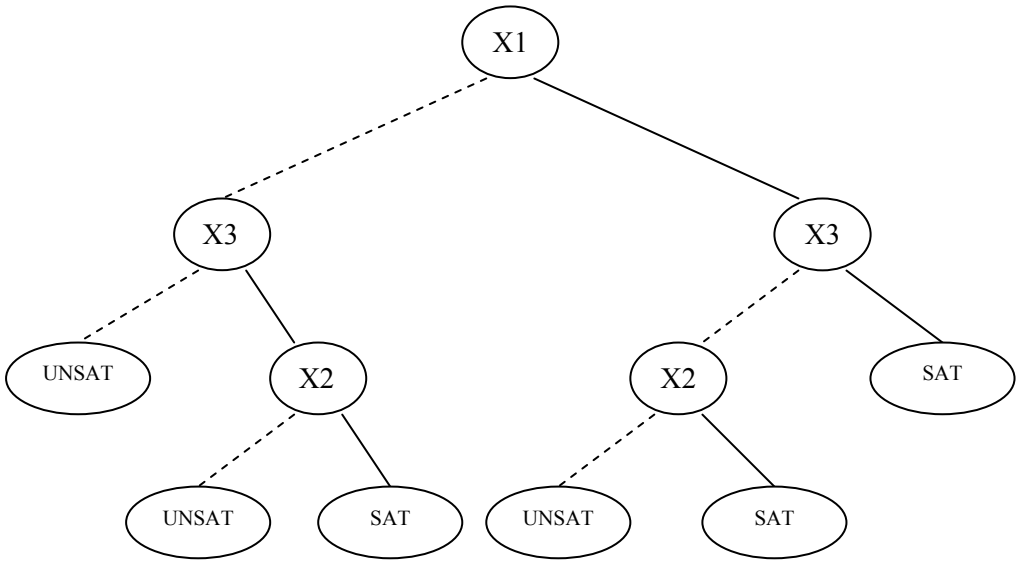


Figure 2.12. Trace d'une procédure DP exhaustive sous forme d'arbre.

Une branche continue (resp. discontinue) désigne le fait d'assigner la variable correspondante à VRAI (resp. FAUX).

La trace d'une procédure DP exhaustive appliquée à une formule permet d'identifier cette formule étant donné qu'il spécifie ses modèles. Cependant, d'un point de vue compilation de connaissances, une trace de recherche mémorisée dans sa forme actuelle n'est pas immédiatement utile car sa taille est proportionnelle à l'effort fourni dans sa génération et répondre à n'importe quelle requête en se basant sur cette représentation revient à refaire toute la recherche une seconde fois. Ce problème peut être résolu en représentant la trace par un DAG au lieu d'un arbre en appliquant les deux règles de réduction suivantes :

1) Les nœuds isomorphiques (nœuds ayant le même label, le même fils droit et le même fils gauche) doivent être confondus.
2) Tout nœud ayant deux fils identiques est éliminé et les arêtes qui se dirigeaient vers lui seront dirigés vers un de ses fils.

Exemple 2.19

En appliquant ces règles de réduction à l'arbre de la figure 2.128 et en renommant SAT (resp. UNSAT) par 1 (resp. 0), on obtient le DAG de la figure 2.13. On constate que ce DAG n'est rien d'autre que la formule CNF Δ sous forme FBDD.

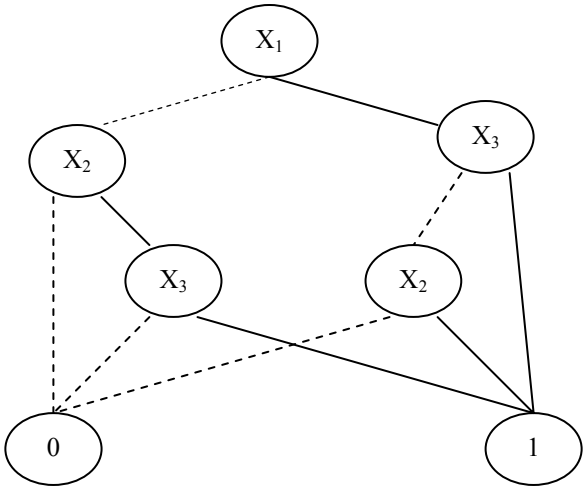


Figure 2.13. Trace d'une procédure DP exhaustive sous forme de DAG.

En se basant sur ce DAG, on peut définir un compilateur FBDD, un compilateur OBDD et un compilateur d-DNNF.

2.6.2. Compilateur FBDD

Afin de mémoriser la trace de la recherche sous forme d'un DAG, on introduit une nouvelle fonction dite *get-node*. La fonction get-node retourne un nœud de décision étiqueté par le premier argument, son fils gauche correspond au second argument et son fils droit au troisième argument. En outre, cette fonction utilise une technique bien connue dans la littérature BDD qu'est la technique de la table des *nœuds uniques* permettant de vérifier les deux règles de réduction décrites précédemment. En effet, tous les nœuds créés par get-node sont mémorisés dans une table et get-node ne crée pas de nœud redondant si :

1) le nœud à créer existe déjà dans la table (ce nœud existant est retourné) ou
2) le second et le troisième arguments sont identiques (un des deux est retourné).

D'autre part, des portions du DAG peuvent être explorées plus d'une fois. La solution réside dans l'utilisation d'un cache. Ainsi, le résultat d'un appel récursif fbdd(Δ) est mémorisé dans ce cache avant d'être retourné en étant indexé par une clef identifiant Δ. Pour calculer une clef , plusieurs méthodes ont été proposées. Le lecteur pourrait se référer à [Darwiche 2002, Darwiche 2004, Huang and Darwiche 2004]. L'algorithme suivant décrit formellement ce compilateur FBDD :

Procédure 2.5 fbdd(Δ)
/* Δ une formule CNF */
début

si il existe une clause vide dans Δ alors **retourner** feuille_0 ;
si il n'y a pas de variable dans Δ alors **retourner** feuille_1 ;
clé ← calculer_clé(Δ) ;
si cache(clé) ≠ null **alors retourner** cache(clé) ;
sélectionner une variable *x* de Δ ;
résultat ← get-node(*x*, fbdd ($\Delta_{(x=0)}$), fbdd($\Delta_{(x=1)}$)
insérer_cache(clé, résultat) ;
retourner résultat ;

fin

2.6.3. Compilateur OBDD

Dans le compilateur fbdd décrit par la procédure 2.5, le choix de la prochaine variable à instancier se fait dynamiquement selon une heuristique de branchement. En substituant cet ordre de sélection dynamique par un ordre statique, on génère une formule OBDD. On obtient ainsi un compilateur obdd ayant comme second argument un ordre de sélection de variables noté π. Cet ordre doit être respecté dans la construction du DAG.

Procédure 2.6 obdd(Δ, π)
/* Δ une formule CNF */
/* π un ordre des variables de Δ */
début

si il existe une clause vide dans Δ alors **retourner** feuille_0 ;
si il n'y a pas de variable dans Δ alors **retourner** feuille_1 ;
clé ← calculer_clé(Δ) ;
si cache(clé) ≠ null **alors retourner** cache(clé) ;
x ← première variable de π qui apparaît dans Δ ;
résultat ← get-node(x, obdd ($\Delta_{(x=0)}$), obdd($\Delta_{(x=1)}$)
insérer_cache(clé, résultat) ;
retourner résultat;

fin

On note que la méthode standard [Bryant 1986] largement utilisée en vérification formelle dans la construction des OBDDs présente le problème suivant : les OBDDs intermédiaires utilisés dans la procédure de construction peuvent être d'une taille colossale ce qui empêche de continuer le calcul même si la taille de l'OBDD final est raisonnable. Par contre, le compilateur obdd qu'on vient de décrire ne souffre pas de cet écueil.

2.6.4. Compilateur d-DNNF

Le langage FBDD étant un sous ensemble du langage d-DNNF, il est possible de définir un compilateur d-DNNF en omettant certaines contraintes au niveau du compilateur FBDD. Précisément, au lieu de se brancher directement sur une variable à instancier, on décompose d'abord la formule en question en des sous-formules disjointes. Ces sous-formules seront compilées récursivement et liées sous forme d'un nœud de conjonction $\wedge$ via la procédure get-and-node. Par conséquent, c'est seulement dans le cas où la formule n'est pas décomposable qu'on se branche sur une variable comme pour la procédure DP classique. Ce compilateur d-DNNF est décrit par la procédure suivante :

Procédure 2.7 d-dnnf(Δ)
début
/* Δ une formule CNF */

si il existe une clause vide dans Δ alors **retourner** feuille_0 ;
si il n'y a pas de variable dans Δ alors **retourner** feuille_1 ;
composantes ← partitions disjointes de Δ ;
si |composantes| > 1 **alors**
début
conjonctions = {}
pour tout Δc ∈ composantes **faire**
conjonctions = conjonctions ∪ {d-dnnf(Δc)}
fait
retourner get-node-and(conjonctions) ;
fin

clé ← calculer_clé(Δ) ;
si cache(clé) ≠ null **alors retourner** cache(clé) ;
sélectionner une variable *x* de Δ ;
résultat ← get-node(x, fbdd ($\Delta_{(x=0)}$), fbdd($\Delta_{(x=1)}$)
insérer_cache(clé, résultat) ;
retourner résultat;

fin

On remarque que ce compilateur fait recours à une décomposition dynamique : les composantes disjointes sont calculées après chaque instanciation d'une variable. Il est possible et surtout moins coûteux de procéder par une décomposition statique et ce en utilisant un arbre de décomposition.

Enfin, il est à noter que la procédure DP présente une limitation qui se trouve dans le fait d'imposer la propriété de déterminisme. En effet, il s'avère qu'il est malheureusement impossible de proposer un compilateur DNNF qui ne soit pas forcément d-DNNF en se basant sur cette procédure.

2.7 Conclusion

Dans ce chapitre, on s'est focalisé sur la compilation logique d'une base de connaissances propositionnelle. On conclut ce chapitre en évoquant brièvement d'autres méthodes de compilation telles que la compilation non logique et la compilation en logique non classique.

Toutes les méthodes de compilation d'une base de connaissances propositionnelle qu'on a présentées jusqu'ici génèrent une formule logique. Cependant, la compilation de connaissances peut générer n'importe quelle structure de données qui ne soit pas forcément une formule logique. L'essentiel est de réduire la complexité de calcul on-line. En particulier, quand le nombre de requêtes possibles est polynomiale, une table des réponses déjà calculées s'avère pratique. Dans [Moses and Tennenholtz 1996], on considère les requêtes qui peuvent être vues comme une conjonction de requêtes de base. Lorsque le nombre de requêtes de base est polynomial par rapport au nombre de variables qui y figurent, on parle de *base efficace*. Pour les requêtes admettant une base efficace, on peut calculer et mémoriser off-line les réponses à ces requêtes de base, et obtenir efficacement on-line la réponse à n'importe quelle requête en consultant la table et en conjoignant les réponses. On note que les requêtes k-CNF admettent une base efficace.

D'autre part, les problèmes de raisonnement en logique non classique sont au-delà des classes NP et CoNP. Dans ce cas, traduire une formule d'une logique non classique vers une formule de la logique classique est aussi une compilation car le raisonnement en logique classique par exemple en logique propositionnelle 'n'est que NP ou CoNP'. Dans cet esprit, on décrit dans [Gelfond *et al*. 1989] une méthode de compilation qui translate une base de connaissances décrite en logique de circonscription vers une base de connaissances exprimée en logique classique.

Chapitre 3

Logique possibiliste

Sommaire

«... la connaissance est une navigation dans un océan d'incertitudes à travers des archipels de certitudes.»
Edgar Morin

3.1. Introduction

En logique classique, une connaissance est soit vraie soit fausse. Néanmoins, les connaissances dont nous disposons sont souvent entachées d'incertitude. Il existe plusieurs modèles pour représenter et traiter l'incertitude. Le modèle le plus fréquemment utilisé est la théorie des probabilités, et en particulier les réseaux bayésiens. Malheureusement, ces derniers ont plusieurs limitations. La principale limitation se trouve dans la nécessité de fournir un nombre considérable de données numériques précises : une contrainte généralement difficile à satisfaire. D'autres modèles ont été élaborés afin de palier cette limitation. Ces derniers remettent en cause ce traitement *quantitatif* et consistent à représenter l'incertitude d'une façon *qualitative* ou *ordinale* ce qui signifie que le traitement de l'incertitude se base sur une relation de *pré-ordre* (c'est-à-dire une relation réflexive et transitive) entre les différentes connaissances. Ce pré-ordre peut être total ou partiel et peut être codé de différentes manières. Parmi les modèles qualitatifs où on considère un pré-ordre total, on peut citer la *logique possibiliste* dont une présentation sommaire fait l'objet de ce chapitre.

La logique possibiliste [Dubois *et al*. 1994] est une extension de la logique classique issue de la théorie des ensembles flous [Zadeh 1965]. Au niveau syntaxique, une formule possibiliste n'est rien d'autre qu'une formule classique qui se voit assigner un poids reflétant son degré de certitude par rapport aux autres formules. Le résultat est appelé *base de croyances possibiliste* où le mot croyance désigne simplement une connaissance incertaine ou plausible. Au niveau sémantique, au lieu de partitionner l'ensemble des interprétations en deux parties (modèles et contre-modèles), on a une partition plus raffinée, dite *distribution de possibilité*, où les contre-modèles des croyances moins plausibles sont préférés aux contre-modèles des croyances plus plausibles.

3.2. Distribution de possibilité

Au niveau sémantique, la notion de base en logique possibiliste est la distribution de possibilité, dénotée par π, qui est une fonction de l'ensemble des interprétations Ω dans l'intervalle [0, 1]. $\pi(\omega)$ représente le degré de cohérence ou de compatibilité de ω avec les croyances disponibles. Par convention, $\pi(\omega) = 1$ veut dire qu'il est totalement possible que ω soit le monde réel, $0 < \pi(\omega) < 1$ signifie qu'il est possible que ω soit le monde réel et enfin $\pi(\omega) = 0$ signifie qu'il est certain que ω n'est pas le monde réel. En plus, $\pi(\omega) > \pi(\omega')$ signifie que ω est plus plausible que ω'. La condition de normalisation exprime le fait qu'il existe une interprétation dans Ω qui soit totalement possible : $\exists\ \omega \in \Omega, \pi(\omega) = 1$.

Les notions de cohérence et d'inférence d'une formule en logique classique deviennent ainsi graduelles et donnent respectivement les mesures de *cohérence* (ou de *possibilités*) et de *certitude* (ou de *nécessité*). En effet, à partir d'une distribution de possibilité π , on définit deux évaluations pour une formule donnée φ : son degré de cohérence et son degré de certitude qu'on définit comme suit :

- Le degré de cohérence d'une formule φ, $\Pi(\varphi) = Max\{\pi(\omega) : \omega \models \varphi\}$, évalue dans quelle mesure φ est cohérente avec les croyances disponibles exprimées par π.
- Le degré de certitude d'une formule φ, $N(\varphi) = 1 - \Pi(\neg\varphi)$, évalue dans quelle mesure φ est déductible à partir des croyances disponibles.

Si on se réfère à la logique classique, $\Pi(\varphi) = 1$ exprime le fait que φ est cohérente avec la base (au sens de la logique classique) et $N(\varphi) = 1$ signifie que φ est déductible à partir de la base. La dualité $N(\varphi) = 1 - \Pi(\neg\varphi)$ étend celle de la logique classique où une formule φ est inférable à partir de la base ($N(\varphi) = 1$) si et seulement si sa négation $\neg\varphi$ est incohérente avec cette base ($\Pi(\neg\varphi) = 0$).

3.3. Bases de croyances possibilistes

La logique possibiliste dans sa version la plus simple appelée *logique possibiliste standard* ou *logique possibiliste évaluée-nécessité* manipule uniquement des formules évaluées certitude. En fait, il est plus facile d'élaborer des algorithmes dans cette version ensuite les généraliser. En outre, d'un point de vue représentation de connaissances, cette version s'avère suffisante pour de nombreuses applications. Par la suite, on se retreint à la logique possibiliste standard.

Au niveau syntaxique, on définit une base de croyances possibiliste Σ par un ensemble de formules pondérées : $\Sigma = \{(p_i, a_i) : i = 1..n\}$ où p_i est une formule classique et a_i appartient à (0, 1]. Le couple (p_i, a_i) signifie que le degré de certitude de p_i est au moins égal à a_i ($N(p_i) \geq a_i$).

D'autre part, les bases de croyances possibilistes peuvent être vues comme des représentations compactes des distributions de possibilité. En effet, on peut associer à toute base possibiliste Σ une distribution de possibilité unique que l'on notera par π_Σ. On rappelle que le degré de possibilité d'une interprétation est son degré de compatibilité ou de cohérence avec la base. Considérons le cas où la base contient une seule formule $\{(p, a)\}$. Si une interprétation ω satisfait la formule p alors ω est complètement cohérente avec la base, par conséquent $\pi_\Sigma(\omega) = 1$. Si ω ne satisfait pas p alors $\pi_\Sigma(\omega)$ doit être tel que plus le degré de certitude a de p est grand, moins ω est possible. En particulier, si p est complètement certaine ($a = 1$) alors ω est complètement impossible ($\pi_\Sigma(\omega)=0$). En général, si la base contient plus d'une formule alors les interprétations satisfaisant toutes les croyances auront le plus grand degré de possibilité qu'est 1 tandis que les autres interprétations seront classées par rapport à la croyance du plus grand poids qu'elles falsifient. De ce fait, on a : $\forall\ \omega \in \Omega$

$$\pi_\Sigma(\omega) = \begin{cases} 1 \text{ si } \forall\ (p_i, a_i) \in \Sigma,\ \omega \models p_i \\ 1 - Max\{a_i : (p_i, a_i) \in \Sigma \text{ et } \omega \not\models p_i\} \text{ sinon.} \end{cases}$$

Comme on peut le voir, $\pi_\Sigma(\omega)$ est élevé lorsque ω n'est pas un contre modèle d'une formule p_i ayant un degré de certitude élevé.

Exemple 3.1
Soient q et r deux propositions qui désignent respectivement « *Les scientifiques ont intercepté des signaux étranges provenant de l'espace extra terrestre.* » et « *Il y'a des extra terrestres dans d'autres planètes.* ». Soit $\Sigma = \{(\neg q \vee r, 0.9), (r, 0.5)\}$.

La distribution de possibilité π_Σ induite par cette base est la suivante :

$$\begin{cases} \pi_\Sigma(\neg q \wedge \neg r) = 1 - 0.5 = 0.5 \\ \pi_\Sigma(\neg q \wedge r) = 1 \\ \pi_\Sigma(q \wedge \neg r) = 1 - 0.9 = 0.1 \\ \pi_\Sigma(q \wedge r) = 1 \end{cases}$$

Les interprétations $(\neg q \wedge r)$ et $(q \wedge r)$ sont totalement possibles : elles sont cohérentes avec la base Σ. L'interprétation $(\neg q \wedge \neg r)$ est plus plausible que l'interprétation $(q \wedge \neg r)$: la plus certaine croyance falsifiée par $(\neg q \wedge \neg r)$ est moins certaine que la plus certaine croyance falsifiée par $(q \wedge \neg r)$.

Il est à noter que l'intervalle [0, 1] ne représente qu'une relation de pré-ordre total entre les interprétations. En fait, une distribution de possibilité peut être représentée simplement sous la forme d'une partition $(E_1 \cup \ldots \cup E_n)$ où E_1 contient les interprétations les plus plausibles et E_n contient les interprétations les moins plausibles. Il est de même, une base de croyances possibiliste peut être représentée sous la forme d'une *base stratifiée* $\Sigma = (S_1 \cup \ldots \cup S_n)$ où S_1 contient les croyances les plus certaines et S_n contient les croyances les moins certaines.

3.4. Inférence en logique possibiliste

Pour des raisons de simplicité, on s'intéresse dans ce qui suit uniquement aux bases de croyances possibilistes propositionnelles. Avant de présenter le processus d'inférence en logique possibiliste, on donne les définitions suivantes :

Définition 3.1 (*a*-coupe)
Soient $\Sigma = \{(p_i, a_i) : i = 1..n\}$ une base de croyances possibiliste et $a \in (0, 1]$. On appelle la ***a*-coupe** de Σ, dénotée par Σ_a, la base propositionnelle définie par : $\Sigma_a = \{ p_i : (p_i, a_i) \in \Sigma \text{ et } a_i \geq a\}$.

Définition 3.2 (degré d'incohérence)
Soit $\Sigma = \{(p_i, a_i) : i = 1..n\}$ une base de croyances possibiliste. Le **degré d'incohérence** de Σ, dénoté par $Inc(\Sigma)$, est défini par : $Inc(\Sigma) = Max\{a_i : \Sigma_{ai} \text{ est incohérente}\}$.

Si Σ est cohérente (c'est-à-dire $\Sigma^* = \{p_i : i = 1..n\}$ est cohérente au sens classique), on pose alors $Inc(\Sigma) = 0$.

En présence d'incohérence, la relation d'inférence classique produit des conclusions triviales : elle permet d'inférer n'importe quelle formule (principe connu sous le nom d'*explosion* ou de *ex falso quodlibet sequitur*). Donc la relation d'inférence classique ne peut être utilisée directement en logique possibiliste. En logique possibiliste, on définit deux relations d'inférence comme suit :

Première forme d'inférence

La première relation d'inférence possibiliste consiste à inférer à partir d'une base de croyances possibiliste les conséquences logiques classiques de la sous base construite de formules dont le degré de certitude est strictement supérieur au degré d'incohérence de la base possibiliste en question. Une définition formelle de cette relation d'inférence est la suivante :

Définition 3.3
Une formule p est conséquence possibiliste de Σ, dénotée par $\Sigma \mid\!=_\pi p$, si et seulement s'il existe un degré $a \in (0, 1]$ tel que $\Sigma_a \models p$ et $a > Inc(\Sigma)$.

Dans ce qui suit, INF_SMPL dénote le problème de décision correspondant à la première forme d'inférence en logique possibiliste.

La condition $a \geq Inc(\Sigma)$ est vérifiée si et seulement si Σ_a est cohérente. Donc ce problème requiert un test de satisfiabilité et un test de non satisfiabilité. La complexité du problème INF_SMPL est donnée par la proposition suivante :

Proposition 3.1
Le problème INF_SMPL est BH_2-complet. [Lang 2000]

Exemple 3.2
Soit $\Sigma = \{(\neg q \vee r, 0.8), (q, 0.6), (s, 0.6), (\neg r, 0.5), (t, 0.3)\}$.

$Inc(\Sigma) = 0.5$.
$\{\neg q \vee r, q\} \models r$, c'est-à-dire il existe $a = 0.6 > Inc(\Sigma)$ tel que $\Sigma_a \models r$. Par conséquent, $\Sigma \models_\pi r$.

Deuxième forme d'inférence

La seconde relation d'inférence en logique possibiliste, qui est plus intéressante, infère les mêmes conséquences que précédemment et, de surcroît, leur associe un degré de certitude. Avant de définir formellement cette forme d'inférence, on donne la définition suivante :

Définition 3.4 (*Val*(*p*, Σ))
Etant données une base possibiliste Σ et une formule *p*. *Val*(*p*, Σ) est donnée par :
$Val(p, \Sigma) = Max\{a_i : \Sigma_{ai} \models p\}$.

Il est à noter que *Val*(*p*, Σ) n'est rien d'autre que *Inc*(Σ ∪ (¬*p*,1)) donc ce calcul peut se faire par dichotomie en utilisant n'importe quel solver SAT. D'où la proposition suivante :

Proposition 3.2
Etant données une base possibiliste Σ et une formule *p*. Le calcul de *Val*(*p*, Σ) requiert ($\log_2 n$) tests de satisfiabilité où *n* est le nombre des différents degrés de certitude dans Σ. [Lang 2000]

On définit maintenant la deuxième forme d'inférence possibiliste :

Définition 3.5
Une formule *p* est une conséquence possibiliste de Σ de degré de certitude *a*, dénotée par $\Sigma \models_\pi (p, a)$, si et seulement si $a = Val(p, \Sigma) > \text{Inc}(\Sigma)$.
Par la suite, INF_DGR dénotera le problème de décision correspondant à la seconde forme d'inférence possibiliste.

Exemple 3.3
Soit $\Sigma = \{(\neg q \vee r, 0.8), (q, 0.6), (s, 0.6), (\neg r, 0.5), (t, 0.3)\}$.
On vérifie aisément que $\Sigma \models_\pi (r, 0.6)$.

Le problème INF_DGR peut être résolu en calculant $a = Val(p, \Sigma)$ et en testant ensuite la cohérence de Σ_a. Sa complexité est donnée par la proposition suivante :

Proposition 3.3
Le problème INF_DGR est dans $\Delta_2^P[O(\log n)]$. [Lang 2000]

3.5. Applications de la logique possibiliste

La logique possibiliste constitue un outil puissant permettant la représentation et le traitement qualitatifs de l'incertitude. Elle peut être également utilisée pour modéliser un ensemble de buts à atteindre munis de niveaux de priorité. Elle permet en outre un traitement simple des règles avec

exceptions (un problème pertinent en Intelligence Artificielle) [Benferhat *et al*. 1997]. Cette logique possibiliste n'est pas dénuée d'intérêt pratique. En fait, elle a de plus en plus des applications dans de nombreux domaines pratiques tels que la sécurité informatique, les systèmes multi-agents ainsi que l'informatique médicale. A titre d'exemple, on esquisse les travaux suivants :

Modélisation des politiques de sécurité en logique possibiliste

Garantir la sécurité des informations d'un système revient à assurer certaines propriétés telles que la confidentialité, l'intégrité et la disponibilité. Pour ce faire, une politique de sécurité exprimant les règles de contrôle d'accès doit être décrite formellement. Une modélisation en logique classique est suffisante en absence de conflits. Néanmoins, les règles de contrôle d'accès présentent souvent des situations exceptionnelles ce qui génère des conflits. Par exemple, un conflit apparaît lorsque l'on autorise un utilisateur à effectuer une action et on même temps on lui interdit de l'exécuter dans une situation exceptionnelle.

La logique possibiliste s'avère commode pour exprimer et traiter un tel problème : dans [Benferhat *et al.* 2003], on propose une modélisation des politiques de sécurité (on s'intéresse de près au domaine de la santé où les dossiers médicaux des patients sont de plus en plus traités automatiquement et par conséquent leur sécurité est de plus en plus importante) basées rôles (RBAC pour Role Based Access Control) dans le cadre de la théorie des possibilités. En gros, le codage possibiliste se résume comme suit : les faits et les règles sans exceptions se codent en associant simplement à chacun d'eux un degré de certitude égale à 1. Quant aux règles avec exceptions, on considère que celles décrivant une situation exceptionnelle sont préférées à celles décrivant une situation générale.

Modèle de négociation basé sur la logique possibiliste

La négociation est considérée comme un problème clef dans le développement de systèmes multi-agents. Elle joue un rôle central dans les dialogues entre des agents qui essayent d'arriver à un accord commun quant au partage d'informations ou de ressources. Le processus de négociation est fondé sur le compromis qui survient en raison des préférences des agents (de leurs buts) : un agent peut être disposé à faire des concessions si cela lui permet d'atteindre un état qui le satisfait au moins potentiellement. Tous les mécanismes de négociation sont basés sur l'échange d'offres. Les agents font des propositions qu'ils trouvent acceptables et répondent aux propositions qui leur sont faites. Dans [Amgoud and Prade 2003], on propose une nouvelle approche où la logique possibiliste est employée pour décrire le processus entier de négociation. Cette logique offre un cadre unifié pour représenter non seulement les états mentaux des agents (y compris des croyances sur l'environnement et des buts avec priorités), mais aussi pour mettre à jour les bases de croyances ou les ensembles de buts, et pour décrire la procédure de décision pour choisir une nouvelle offre.

Un autre exemple d'application de la logique possibiliste peut être le projet de développement d'une approche possibiliste du traitement de l'information cérébrale qui s'inscrit dans le cadre d'une collaboration entre l'INSERM (Institut National de la Santé et de la Recherche Médicale) et l'IRIT (Institut de Recherche en Informatique de Toulouse).

Toutefois, on a vu précédemment que le raisonnement en logique possibiliste est onéreux d'un point de vue calculatoire : alors que INF-SMPL est BH_2-complet, INF-DGR est dans $\Delta_2^P[O(\log n)]$.

En logique classique, la compilation de connaissances reste une approche prometteuse afin de palier aux problèmes de complexité calculatoire du raisonnement. L'idée alors est d'utiliser cette approche dans le cadre de la logique possibiliste pour permettre un raisonnement plus efficace.

3.6. Compilateur possibiliste DNF

Le premier compilateur (à notre connaissances) de bases de croyances possibilistes a été dernièrement proposé dans [Benferhat and Prade 2006]. Ce compilateur s'inspire de la méthode présentée dans [Benferhat and Prade 2005] dans le cadre du traitement d'une extension de la logique possibiliste où on définit une relation de pré-ordre partiel entre les degrés de certitude. Les idées sous-jacentes consistent à coder la base possibiliste à compiler Σ sous la forme d'une base propositionnelle K_Σ, à utiliser les notions *d'oubli* de variables et de littéraux et surtout le langage DNF.

Ce compilateur retourne deux bases compilées, dénotées par Comp(Σ) et Comp'(Σ), afin de traiter les problèmes INF_SMPL et INF_DGR respectivement. L'algorithme ASK de la définition 2.2 d'un problème compilable associé à Comp(Σ) n'est rien d'autre que l'inférence classique c'est-à-dire Comp(Σ) est telle que pour toute formule p, $\Sigma \mid\sim_\pi p$ si et seulement si Comp(Σ) $\mid\sim p$. Quant à Comp'(Σ), on lui fait correspondre un autre algorithme tel que pour toute formule p, $\Sigma \mid\sim_\pi (p, a)$ si et seulement si cet algorithme le décide en utilisant Comp'(Σ). Avant de décrire comment opère ce compilateur, on présente le codage propositionnel d'une base de croyances possibiliste et les opérations d'oubli de variable et de littéraux.

3.6.1. Codage propositionnel d'une base possibiliste

Soit $\Sigma = \{(p_i, a_i) : i = 1..n\}$ une base de croyances possibiliste où les formules p_i sont construites à partir d'un ensemble de variables V ($p_i \in PROP_V$). On suppose aussi que $1 \geq a_1 \geq a_2 \geq \ldots \geq a_n > 0$.

La première étape du codage propositionnel de la base possibiliste Σ consiste à associer à chaque degré a_i dans Σ une nouvelle variable propositionnelle que l'on notera par la lettre majuscule correspondante A_i. S dénote l'ensemble des nouvelles variables propositionnelles correspondant aux différents degrés de certitude de Σ. On note que $S \cap V = \emptyset$.

Une formule possibiliste (p_i, a_i) lui est associée la formule propositionnelle $p_i \vee A_i$ où A_i signifie « la situation est A_i-anormale ». Donc $p_i \vee A_i$ se traduit par : « p_i est vrai ou la situation est A_i-anormale ».

On a besoin aussi de traduire la relation d'ordre entre les poids. En effet, l'inégalité $a_i \geq a_j$ est exprimée par le biais de l'implication matérielle $A_i \Rightarrow A_j$ c'est-à-dire par la clause binaire $\neg A_i \vee A_j$. Cette traduction peut être lue comme suit : « si la situation est au moins très anormale (A_i), alors est au moins anormale (A_j). En effet, plus a_i est proche de 1, plus p_i est certaine dans (p_i, a_i) et plus est

exceptionnelle une situation où p_i est fausse. La définition suivante décrit formellement le codage d'une base de croyances possibiliste sous la forme d'une base propositionnelle classique.

Définition 3.6 (Codage propositionnel)
Soit $\Sigma = \{(p_i, a_i) : i = 1..n\}$ une base de croyances possibiliste. Pour tout i, on associe au degré de certitude a_i, une variable propositionnelle A_i. Alors la base propositionnelle associée à Σ, notée par K_Σ, est définie par :
$K_\Sigma = \{ p_i \vee A_i : (p_i, a_i) \in \Sigma \} \cup \{\neg A_i \vee A_{i+1} : i = 1..n\text{-}1 \}$.

On constate que le nombre des nouvelles variables propositionnelles introduites, $| S |$, est égal au nombre de degrés de certitude dans Σ c'est-à-dire n. En outre, le nombre de clauses binaires rajoutées est égal à $n - 1$. On déduit alors que le coût de ce codage n'est pas important.

3.6.2. Oubli de variables et de littéraux

L'oubli de variable [Lin and Reiter 1994] (connu sous d'autres appellations telles que la marginalisation) est une notion importante ayant diverses applications en intelligence Artificielle. Par exemple, étant donnée une formule propositionnelle K, on peut être intéressé uniquement par des requêtes qui ne mentionnent aucune variable de A (A étant un sous ensemble de V). Dans ce cas, il est judicieux de simplifier K et ce en oubliant les variables de A tout en préservant un maximum de connaissances sur les variables non oubliées. On peut citer une autre application dans le cadre du raisonnement en présence d'incohérence. En fait, l'incohérence d'une base peut être due à un excès d'information sur certaines variables qui peuvent ne pas être cruciales. L'idée alors est d'oublier tout ce qui concerne ces variables « responsables de l'incohérence » et parvenir ainsi à restaurer la cohérence.

Définition 3.7 (Oubli de variables)
Soit K une formule propositionnelle. Soit v une variable propositionnelle. L'oubli de v dans K, noté par *ForgetVariable* (K, v), est défini comme suit :
ForgetVariable(K, v) = $K_{v=\bot} \vee K_{v=\top}$ tel que $K_{v=\bot}$ (resp. $K_{v=\top}$) est la formule obtenue à partir de K en remplaçant v par vrai (resp. faux).

Exemple 3.4
Soit $K = (\neg a \vee b) \wedge (a \vee c)$. *ForgetVariable*($K$, a) = $b \vee c$.

Oublier un ensemble de variables dans une formule se fait variable par variable. Soit $A \subseteq V$, oublier A dans une formule propositionnelle K se fait par induction comme suit :
ForgetVariable (K, A) = *ForgetVariable*(*ForgetVariable*(K, v), A-{v}).

Parmi les propriétés de l'oubli de variables on cite les suivantes :

Propriétés de l'oubli de variables

1. *ForgetVariable*($p \vee q$, A) = *ForgetVariable*(p, A) $\vee$ *ForgetVariable*(q, A)
2. Si p ne contient aucune variable de A alors *ForgetVariable* ($p \wedge q$, A) = p $\wedge$ *ForgetVariable* (q, A).

3. Si p est un terme consistent alors *ForgetVariable*(p, A) revient à supprimer les variables de A dans p.
4. Si p est une formule consistante (resp. inconsistante), alors oublier toutes ses variables donne $\top$ (resp. $\bot$).
5. *ForgetVariable*(p, A) est la plus forte conséquence logique de p qui soit indépendante de A.

Quant à l'oubli de littéraux, c'est une notion qui a été récemment introduite dans [Lang *et al*. 2003]. Il s'agit d'une généralisation de l'oubli de variables définie comme suit :

Définition 3.8 (Oubli de littéraux)
Soit K une formule propositionnelle. Soit l un littéral. L'oubli de l dans K, dénoté par *ForgetLiteral*(K, l), est défini comme suit :
ForgetLiteral(K, l) = $K_{l=\top} \vee \neg l \wedge K$.

Exemple 3.5
Soit $K = (\neg a \vee b) \wedge (a \vee c)$. *ForgetLiteral*($K$, $\neg a$) = $c \vee (a \wedge b)$.

Quelques propriétés de l'oubli de littéraux sont les suivantes :

Propriétés de l'oubli de littéraux

1. *ForgetLiteral*($p \vee q$, A) = *ForgetVariable*(p, A) $\vee$ *ForgetVariable*(q, A).
2. Si p est un terme consistent alors *ForgetLiteral*(p, A) revient à supprimer les littéraux de A dans p.
3. *ForgetLiteral* (p, A) est la plus forte conséquence de p qui soit indépendante des littéraux de A.

En général, l'oubli de variables est une opération coûteuse. Toutefois, il s'effectue en temps polynomial si l'on considère des formules DNF [Darwiche and Marquis 2002]. En effet, oublier une variable dans une formule DNF revient à l'oublier dans chaque terme, et l'oublier dans un terme revient simplement à sa suppression de ce dernier.

3.6.3. Description du compilateur

On montre dans un premier temps comment calculer le degré d'incohérence d'une base de croyances possibiliste Σ et ce en utilisant la base propositionnelle correspondante K_Σ et l'oubli de variables et de littéraux. On distingue deux cas : Σ cohérente et Σ incohérente à un degré a_i. Ces deux cas sont traités par les propositions 3.4 et 3.5 respectivement.

Proposition 3.4
Soient $\Sigma = \{(p_i, a_i) : i = 1..n\}$ une base de croyances possibiliste et K_Σ la base propositionnelle qui lui est associée selon la définition 3.6. Soit *NegS* l'ensemble des littéraux négatifs de K_Σ construits à partir de S. Soit F = *ForgetVariable*(K_Σ, V). Alors Σ est cohérente si et seulement si *ForgetLiteral*(F, *NegS*) est une tautologie.

Proposition 3.5
Soient Σ une base de croyances possibiliste et K_Σ la base propositionnelle qui lui est associée selon la définition 3.6. Soit *NegS* l'ensemble des littéraux négatifs de K_Σ construits à partir de *S*. Soit F = *ForgetVariable*(K_Σ, *V*). Alors Σ est inconsistant à un degré a_i si et seulement si *ForgetLiteral*(*F*, *NegS*) $\equiv A_i \wedge \ldots \wedge A_n$.

Une fois qu'on a calculé le degré d'incohérence de Σ, on peut calculer Comp(Σ) (la base compilée utilisée pour INF_SMPL) en utilisant les deux propositions suivantes :

Proposition 3.6
Soient Σ une base de croyances possibiliste et K_Σ la base propositionnelle qui lui est associée selon la définition 3.6. Si Σ est consistante alors :
Comp(Σ) = *ForgetVariable*($\neg A_1 \wedge \ldots \wedge \neg A_n \wedge K_\Sigma$, *S*)

Proposition 3.7
Soient Σ une base de croyances possibiliste et K_Σ la base propositionnelle qui lui est associée selon la définition 3.6. Si Σ est inconsistante à un degré a_i alors :
Comp(Σ) = *ForgetVariable*($\neg A_1 \wedge \ldots \wedge \neg A_{i-1} \wedge K_\Sigma$, *S*).

On montre que compiler dans un premier temps la base propositionnelle K_Σ vers le langage DNF c'est-à-dire DNF(K_Σ) permet d'effectuer le calcul du degré d'incohérence et le calcul de Comp(Σ) en temps polynomial. En outre, Comp(Σ) est à son tour sous forme DNF ce qui permet une inférence classique polynomiale.

Quant à Comp'(Σ) (la base compilée considérée dans le cas de INF_DGR), ce n'est rien d'autre que DNF(K_Σ). Avant de décrire l'algorithme correspondant à Comp'(Σ), on présente formellement le compilateur DNF via l'algorithme suivant :

Algorithme 3.1 : Compilateur possibiliste DNF
Données : une base de croyances possibiliste $\Sigma = \{(p_i, a_i) : i = 1..n\}$
Résultat : une base propositionnelle Comp(Σ)
une base propositionnelle Comp'(Σ)
début

1 : Transformer Σ en K_Σ
2 : Compiler K_Σ vers le langage DNF
3 : Oublier les variables de *V* dans K_Σ
4 : Oublier les littéraux négatifs de *S* dans K_Σ
5 : Calculer le degré d'incohérence a_i
6 : si Σ est cohérente **alors** Comp(Σ) = *ForgetVariable*($\neg A_1 \wedge \ldots \wedge \neg A_n \wedge$ DNF(K_Σ), *S*)
sinon Comp(Σ) = *ForgetVariable*($\neg A_1 \wedge \ldots \wedge \neg A_{i-1} \wedge$ DNF(K_Σ), *S*)
7 : Comp'(Σ) = DNF(K_Σ) ;
Retourner Comp(Σ), Comp'(Σ) ;

fin

L'algorithme correspondant à Comp'(Σ) se base sur la proposition suivante :

Proposition 3.8
Soit p une formule propositionnelle qui est conséquence de Comp(Σ), alors $\Sigma \mid\sim_{\pi} (p, a_i)$ si et seulement si :

p découle logiquement de $ForgetVariable(\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge K_\Sigma, S)$ et
p ne découle pas logiquement de $ForgetVariable(\neg A_1 \wedge \ldots \wedge \neg A_j \wedge A_{j+1} \wedge \ldots \wedge A_n \wedge K_\Sigma, S)$ $\forall$ j < i.

De plus, on montre qu'utiliser K_Σ sous forme DNF c'est-à-dire Comp'(Σ) permet d'effectuer en temps polynomial l'algorithme 3.2 défini comme suit :

Algorithme 3.2 : Calcul du degré de certitude associé à une formule
Données : Comp'(Σ)
une clause p qui est une conséquence de Comp(Σ)
Résultat : un degré de certitude associé à p
début

1 : $k \leftarrow 0$
2 : $X \leftarrow \{A1, \ldots, A_n\}$
3 : tant que ForgetVariable($X \cup$ Comp'(Σ), S) $\not\models p$ **faire**
4 : $k \leftarrow k + 1$
5 : $X \leftarrow X - \{A_k\}$
6 : $X \leftarrow X \cup \{\neg A_k\}$
fait
retourner A_k

fin

3.7. Conclusion

Dans ce chapitre, on a présenté les ingrédients de base de la logique possibiliste à savoir la distribution de possibilité et les bases de croyances possibilistes. On a défini aussi les deux relations d'inférence en logique possibiliste tout en donnant leur complexité computationnelle. On a cité également quelques exemples tangibles de son application. Enfin, on s'est focalisé sur le compilateur possibiliste DNF proposé récemment dans [Benferhat and Prade 2006]. Ce compilateur retourne deux basses propositionnelles qui permettent de répondre aux problèmes INF_SMPL (BH_2-complet) et INF_DGR (dans $\Delta_2^{\mathrm{P}}[O(\log n)]$) en temps polynomial par rapport à la taille de ces bases. Toutefois, en utilisant le langage DNF, cette taille peut être prohibitive et l'utilisation d'autres langages cibles de compilation plus concis tels que DNNF semble judicieuse dans l'optique d'introduire d'autres compilateurs possibilistes plus efficaces.

Chapitre 4

Compilation de bases de croyances possibilistes

Sommaire

« Tout ce que j'ai publié n'est que des fragments d'une grande confession. »
Goethe

4.1. Introduction

La logique possibiliste constitue un formalisme intéressant permettant de modéliser de nombreux problèmes en Intelligence Artificielle. Elle permet de représenter et de traiter des connaissances incertaines, des buts avec priorité ainsi que des règles avec exceptions. Cette logique est de plus en plus utilisée en pratique à titre d'exemple en sécurité informatique, dans les systèmes multi-agents et en informatique médicale. Néanmoins, d'un point de vue calculatoire, on se heurte au problème de complexité du raisonnement possibiliste qui est plus coûteux que le raisonnement propositionnel classique : alors que la première relation d'inférence possibiliste constitue un problème BH_2-complet, la seconde (la plus fréquemment utilisée) est dans $\Delta_2^{\mathrm{P}}[O(\log n)]$. Une solution envisageable est la compilation : une approche qui a atteint un certain stade de maturité en logique propositionnelle où elle s'est avérée prometteuse.

Nous avons présenté à la fin du chapitre précédent une méthode de compilation de bases de croyances possibilistes récemment proposée dans [Benferhat and Prade 2006]. Cette méthode repose principalement sur un codage propositionnel de la base possibiliste en question et sur l'utilisation des opérations d'oubli de variables et de littéraux. La clef de succès de cette méthode réside dans l'utilisation du langage DNF qui supporte l'opération d'oubli et qui permet aussi une inférence classique polynomiale. Le revers de la médaille est que le langage DNF présente une limitation qui se trouve dans le fait qu'il existe des formules logiques qui n'admettent pas de représentation DNF de taille raisonnable ou polynomiale, par contre une telle représentation peut exister en considérant d'autres langages.

Récemment, en compilation de bases de connaissances propositionnelles, et comme nous l'avons vu dans le second chapitre, on a défini un répertoire considérable de langages cibles de compilation. On a également établi une relation de préférence entre ces derniers en fonction de leur concision et du type d'opérations logiques qu'ils permettent d'effectuer en temps polynomial. Il est pertinent alors de tirer profit de ces langages et d'ambitionner de proposer, en se basant sur la méthode de [Benferhat and Prade 2006], de nouveaux compilateurs possibilistes qui soient plus efficaces. Une telle étude, qui représente notre contribution principale, fait l'objet de ce chapitre.

4.2 Développement d'un compilateur possibiliste DNNF

Le langage DNNF est plus concis que le langage DNF. Cette propriété se traduit par le fait qu'une formule propositionnelle peut avoir une formule DNNF équivalente de taille polynomiale. Par contre, cette même formule n'admet que des formules DNF équivalentes de taille exponentielle. De plus, le langage DNNF est plus concis que tous les autres langages cibles de compilation excepté PI : on sait que PI n'est pas plus concis que DNNF mais on ignore l'inverse. Hormis le critère de concision, DNNF présente d'autres avantages en permettant d'effectuer en temps polynomial des opérations logiques qui sont en général suffisantes pour réaliser des applications complexes en Intelligence Artificielle. Par conséquent, il est tout à fait naturel que nous nous focalisions dans un premier temps sur ce langage en vue d'introduire un compilateur DNNF dans le cadre de la logique possibiliste.

Intéressons nous en premier lieu à l'algorithme 3.1 décrivant le compilateur possibiliste DNF proposé par [Benferhat and Prade 2006]. Intuitivement, afin d'avoir un compilateur DNNF, nous proposons de compiler le codage propositionnel K_Σ vers le langage DNNF au lieu de le compiler vers le langage DNF. Nous obtenons alors l'algorithme 4.1. Analysons sa complexité.

D'après la proposition 2.3, nous savons que le langage DNNF satisfait l'oubli de variables. Autrement dit, le langage DNNF permet d'effectuer l'oubli de variables en temps polynomial. En plus, le résultat de cette opération appartient au langage DNNF. En fait, oublier un ensemble de variables A dans une formule propositionnelle K revient simplement à remplacer tous les littéraux construits sur A (c'est-à-dire de la forme v ou $\neg v$ tel que $v \in A$) par Vrai [Darwiche 2001 a].

Exemple 4.1
Soit la formule DNNF suivante : $K = (\neg A \wedge \neg B) \vee (C \wedge B)$.
ForgetVariable(K, $\{A, C\}$) = (Vrai $\wedge \neg B$) $\vee$ (Vrai $\wedge B$) = $\top$.

Algorithme 4.1
Données : une base de croyances possibiliste $\Sigma = \{(p_i, a_i) : i = 1..n\}$
Résultat : une base propositionnelle Comp(Σ)
une base propositionnelle Comp'(Σ)
début

1 : Transformer Σ en K_Σ
2 : Compiler K_Σ vers le langage DNNF
3 : Oublier les variables de V dans K_Σ
4 : ***Oublier les littéraux*** négatifs de S dans K_Σ
5 : Calculer le degré d'incohérence a_i
6 : si Σ est cohérente **alors** Comp(Σ) = ***ForgetVariable*($\neg A_1 \wedge \ldots \wedge \neg A_n \wedge$ *DNNF*(K_Σ), *S*)**
sinon Comp(Σ) = ***ForgetVariable*($\neg A_1 \wedge \ldots \wedge \neg A_{i-1} \wedge$ *DNNF*(K_Σ), *S*)**
7 : Comp'(Σ) = DNNF(K_Σ)
retourner Comp(Σ), Comp'(Σ) ;

fin

Nous déduisons alors que l'étape 3 s'effectue en temps polynomial et génère une formule DNNF. Toutefois, et contrairement au langage DNF qui permet d'effectuer l'oubli de littéraux en temps polynomial, nous ignorons si le langage DNNF le fait ou non. De ce fait, rien ne nous garantit que l'étape 4 s'effectue en temps polynomial en considérant le langage DNNF.

D'autre part, pour tout $i = 1 .. n$: $\neg A_1 \wedge \ldots \wedge \neg A_i \wedge$ DNNF(K_Σ) n'est pas une formule DNNF car $Var(\neg A_1 \wedge \ldots \wedge \neg A_i) \cap Var(DNNF(K_\Sigma)) \neq \varnothing$ donc l'oubli de variables à l'étape 6 ne se fait pas en temps polynomial. En outre, Comp(Σ) n'appartient pas au langage DNNF (ni à un autre langage cible de compilation). Par conséquent, l'inférence classique à partir de Comp(Σ) n'est pas polynomiale.

Intéressons nous maintenant à l'algorithme associé à Comp'(Σ) pour le calcul du degré de certitude d'une formule où Comp'(Σ) = DNNF(K_Σ) (voir algorithme 4.2). Le même problème que précédemment ce pose au niveau de l'étape 3 de cet algorithme : $X \cup Comp'(\Sigma) = X \cup DNNF(K_\Sigma)$ n'est pas une formule DNNF car $Var(X) \cap Var(DNNF(K_\Sigma)) \neq \varnothing$ donc ni l'oubli de variables ni l'inférence de p ne peuvent être effectués en temps polynomial. En plus dans ce cas, il s'agit d'un traitement itératif et qui se fait on-line.

Dans ce qui suit, nous allons essayer de résoudre graduellement ces problèmes et ce en prouvant dans un premier temps que DNNF satisfait l'oubli de littéraux et voir d'un autre côté comment préserver la propriété de décomposabilité tout au long des algorithmes 4.1 et 4.2 afin d'effectuer en temps polynomial les opérations d'oubli de variables et la déduction et ainsi résoudre en temps polynomial les problèmes INF_SMP et INF_DGR en utilisant le langage DNNF.

Algorithme 4.2
Données : Comp'(Σ)
une clause *p* qui est une conséquence possibiliste de Σ
Résultat : le degré de certitude associé à *p*
début

1 : $k \leftarrow 0$
2 : $X \leftarrow \{A1, \ldots, A_n\}$
3 : tant que ***ForgetVariable(X ∪ Comp'(Σ), S) |≠ p*** **faire**
4 : $k \leftarrow k + 1$
5 : $X \leftarrow X - \{A_k\}$
6 : $X \leftarrow X \cup \{\neg A_k\}$
fait
retourner A_k

fin

4.2.1. DNNF et l'oubli de littéraux

L'oubli de littéraux est une généralisation de l'oubli de variables et donc dire que DNNF satisfait l'oubli de littéraux (ou non) nécessite une démonstration et c'est ce que nous allons faire dans cette section.

Proposition 4.1
Soient *p* et *q* deux formules propositionnelles. Soit *L* un ensemble de littéraux. Si Var(*p*) ∩ Var(*q*) = Ø alors on montre que :
ForgetLiteral($p \wedge q$, *L*) = *ForgetLiteral*(*p*, *L*) ∧ *ForgetLiteral*(*q*, *L*).

Preuve
On raisonne par récurrence sur la taille de *L*, | *L* |:

- Pour | *L* | = 0,

ForgetLiteral($p \wedge q$, Ø) = $p \wedge q$ = *ForgetLiteral*(*p*, Ø) ∧ *ForgetLiteral*(*q*, Ø).

- On suppose maintenant que c'est vrai pour tout *L'* tel que | *L* '| ≤ *k* et on le montre pour *L* tel que | *L* | = *k*+1.

On pose alors $L = \{l_1, \ldots, l_{k+1}\}$.

ForgetLiteral($p \wedge q$, $\{l_1, \ldots, l_{k+1}\}$) ≡ *ForgetLiteral* (*ForgetLiteral* ($p \wedge q$, $\{l_1, \ldots, l_k\}$), l_{k+1})
{Par définition.}

≡ *ForgetLiteral* (*ForgetLiteral* (*p*, $\{l_1, \ldots, l_k\}$) ∧ *ForgetLiteral* (*q*, $\{l_1, \ldots, l_k\}$), l_{k+1})
{Par hypothèse de récurrence.}

On distingue deux cas:

✓ **<u>Cas 1</u> : ($l_{k+1} \in p$) ou ($\neg l_{k+1} \in p$)**

On déduit que ($l_{k+1} \notin q$) et ($\neg l_{k+1} \notin q$) car Var(p) ∩ Var(q) = ∅

Donc ($l_{k+1} \notin$ *ForgetLiteral* (q, $\{l_1,\ldots, l_k\}$) et ($\neg l_{k+1} \notin$ *ForgetLiteral* (q, $\{l_1,\ldots, l_k\}$)

On pose (pour simplifier l'écriture) :

p' = *ForgetLiteral* (p, $\{l_1,\ldots, l_k\}$) ;
q'= *ForgetLiteral* (q, $\{l_1,\ldots, l_k\}$) et
$l = l_{k+1}$.

Donc *ForgetLiteral*($p' \wedge q'$, l)
$= [(p' \wedge q')_{(l=\top)}] \vee [\neg l \wedge (p' \wedge q')]$ {Par définition.}

$= [(p'_{(l=\top)} \wedge q'_{(l=\top)}] \vee [\neg l \wedge (p' \wedge q')]$

$= [(p'_{(l=\top)} \wedge q'] \vee [(\neg l \wedge p') \wedge q')]$ {Car $l \notin q'$ et $\neg l \notin q'$ et donc $q' = q'_{(l=\top)} = q'_{(l=\bot)}$ }

$= q' \wedge [p'_{(l=\top)} \vee (\neg l \wedge p')]$

$= q' \wedge$ *ForgetLiteral*(p', l)

D'autre part
$q' \equiv q' \vee (\neg l \wedge q')$
$\equiv q'_{(l=\top)} \vee (\neg l \wedge q')$ {Car $l \notin q'$ et $\neg l \notin q'$ et donc $q' = q'_{(l=\top)} = q'_{(l=\bot)}$ }
$\equiv$ *ForgetLiteral*(q', l) (Par définition.)

Donc :
ForgetLiteral($p' \wedge q'$, l) ≡ *ForgetLiteral* (p', l) ∧ *ForgetLiteral*(p', l)

C'est à dire
ForgetLiteral($p \wedge q$, $\{l_1,\ldots, l_{k+1}\}$) ≡
ForgetLiteral(*ForgetLiteral* (p, $\{l_1,\ldots, l_k\}$), l_{k+1}) ∧ *ForgetLiteral*(*ForgetLiteral* (q, $\{l_1,\ldots, l_k\}$), l_{k+1})

≡ *ForgetLiteral*(p, $\{l_1,\ldots, l_{k+1}\}$) ∧ *ForgetLiteral*(q, $\{l_1,\ldots, l_{k+1}\}$) {Par définition}

✓ **<u>Cas 2</u> : ($l_{k+1} \in q$) ou ($\neg l_{k+1} \in q$)**

On raisonne de la même manière que pour le premier cas. □

En utilisant la proposition 4.1, on montre que le langage DNNF permet d'effectuer l'oubli de littéraux en temps polynomial en plus, le résultat de cette opération est dans DNNF.

Proposition 4.2
Le langage DNNF satisfait d'oubli de littéraux.

Preuve
Soit L un ensemble de littéraux.
On raisonne par induction sur la structure d'une formule DNNF.

Si l est un littéral alors l est une formule DNNF
Si p et q sont deux formules DNNF alors $p \vee q$ est une formule DNNF
Si p et q sont deux formules DNNF et de plus $\text{Var}(p) \cap \text{Var}(q) = \varnothing$ alors $p \wedge q$ est une formule DNNF.

En parallèle, on a :

1. $ForgetLiteral(l, L) \equiv \begin{cases} l \text{ si } l \notin L \\ \text{Vrai si } l \in L \end{cases}$
2. $ForgetLiteral(p \vee q, L) \equiv \text{ForgetLiteral}(p, L) \vee \text{ForgetLiteral}(q, L)$ (propriété 1 de l'oubli de littéraux).
3. $ForgetLiteral(p \wedge q, L) \equiv \text{ForgetLiteral}(p, L) \wedge \text{ForgetLiteral}(q, L)$ si $\text{Var(p)} \cap \text{Var(q)} = \varnothing$ (Proposition 4.1)

Donc oublier les littéraux de L dans une formule DNNF K revient simplement à remplacer par Vrai les littéraux de L qui apparaissent dans K. Il est clair que ce traitement s'effectue en temps polynomial par rapport à la taille de K. En plus le résultat est DNNF. □

Exemple 4.2
Soit la formule DNNF suivante : $K = (\neg A \wedge \neg B) \vee (C \wedge B)$.
$\text{ForgetLiteral}(K, \{A, C\}) = (\neg A \wedge \neg B) \vee (\text{Vrai} \wedge B) = (\neg A \wedge \neg B) \vee B$.

4.2.2. Calcul du degré d'incohérence

Afin de calculer le degré d'incohérence, nous procédons de la même manière présentée dans [Benferhat and Prade 2006] : on vérifie pour $i = 1$ à n si $ForgetLiteral\ (F, NegS)$ est équivalente à $A_i \wedge \ldots \wedge A_n$ avec $F = ForgetVariable(K_\Sigma, V)$ et $NegS$ l'ensemble des littéraux négatifs de K_Σ construits à partir de S. On note qu'en général, le nombre de niveaux n n'est pas important ce qui permet d'effectuer efficacement ces tests d'équivalences.

Une autre méthode dédiée au calcul du degré d'incohérence très proche de celle ci a été présentée dans [Benferhat and Prade 2005]. La différence est que cette dernière ne fait pas recours à des tests d'équivalence comme le montre la proposition suivante :

Proposition 4.3
Soit δ(Σ) une formule propositionnelle obtenue à partir de *Inc*(Σ) en remplaçant le maximum par une conjonction, le minimum par une disjonction et les degrés de certitude par les variables propositionnelles correspondantes. On a alors δ(Σ) est équivalente à la plus forte conséquence logique de K_Σ qui soit indépendante des variables de V et des littéraux négatifs de *S* c'est-à-dire équivalent à *ForgetLiteral* (*F*, *NegS*) avec *F* = *ForgetVariable*(K_Σ, *V*).

Autrement dit, une fois que nous calculons *ForgetLiteral* (*F*, *NegS*) tel que *F* = *ForgetVariable*(K_Σ, *V*), nous remplaçons l'opérateur de conjonction par la fonction maximum, l'opérateur de disjonction par la fonction minimum et les variables propositionnelles de *S* par les degrés de certitude correspondants puis nous évaluons l'expression arithmétique résultante. Nous remarquons que ce calcul se fait en temps polynomial indépendamment du nombre de degrés de certitude employés dans la base.

4.2.3. Préserver la propriété de décomposabilité

Afin de palier le problème de non décomposabilité qui survient à l'étape 6 de l'algorithme 4.1, nous proposons une autre méthode pour le calcul de Comp(Σ) qui ne souffre pas de ce problème. Avant cela, nous montrons dans un premier temps que le codage propositionnel K_Σ d'une base de croyances possibiliste Σ admet une forme équivalente spécifique qui nous sera utile par la suite. Nous rappelons que le codage propositionnel K_Σ d'une base de croyances possibiliste Σ proposé par [Benferhat and Prade 2006], donné formellement par la définition 3.6, consiste à associer à chaque degré de certitude dans Σ une nouvelle variable propositionnelle dénotée par la lettre majuscule correspondante, à traduire chaque formule possibiliste (p, a) par la formule propositionnelle classique $p \vee A$ et à exprimer la relation $a \geq b$ par la clause binaire $\neg A \vee B$.

Exemple 4.3
Soit Σ = {(p, 0.9), (q, 0.7), (r, 0.5)} une base de croyances possibiliste.
Soient A_1, A_2 et A_3 trois variables propositionnelles associées aux degrés de certitude 0.9, 0.7 et 0.5 respectivement.
D'après la définition 3.6, la base propositionnelle associée est :
$K_\Sigma = \{p \vee A_1, q \vee A_2, r \vee A_3, \neg A_1 \vee A_2, \neg A_2 \vee A_3\}$.

Dans la proposition suivante, nous décrivons une forme particulière qu'admet le codage propositionnel K_Σ :

Proposition 4.4
Soient $\Sigma = \{(p_i, a_i) : i = 1..n\}$ une base de croyances possibiliste et K_Σ la base de connaissances propositionnelle correspondante selon la définition 3.6. On montre alors que K_Σ est équivalente à :
$(A_1 \wedge \ldots \wedge A_n) \vee (\neg A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge p_1) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge p_1 \wedge \ldots \wedge p_n)$.

Preuve
Raisonnons par récurrence par rapport au nombre de niveaux *n*.

* On commence par le cas de base : $n = 1$

$K_\Sigma \equiv A_1 \vee p_1$ (par définition)
$\equiv A_1 \vee (\neg A_1 \wedge p_1)$. Donc c'est vérifié pour le cas de base.

* On suppose maintenant que c'est vrai jusqu'à l'ordre n et on le montre pour n+1 :

Soit Σ une base de connaissances possibiliste de n+1 niveaux, donc :
$K_\Sigma \equiv (A_1 \vee p_1) \wedge \ldots \wedge (A_n \vee p_n) \wedge (A_{n+1} \vee p_{n+1}) \wedge (\neg A_1 \vee A_2) \wedge \ldots \wedge (\neg A_{n-1} \vee A_n) \wedge (\neg A_n \vee A_{n+1})$
{ Par definition.}
$\equiv [(A_1 \vee p_1) \wedge \ldots \wedge (A_n \vee p_n) \wedge (\neg A_1 \vee A_2) \wedge \ldots \wedge (\neg A_{n-1} \vee A_n)] \wedge (A_{n+1} \vee p_{n+1}) \wedge (\neg A_n \vee A_{n+1})$

On utilise à ce stade l'hypothèse de la récurrence et le fait que $(A_{n+1} \vee p_{n+1}) \wedge (\neg A_n \vee A_{n+1}) \equiv A_{n+1} \vee (\neg A_n \wedge p_{n+1})$.

$K_\Sigma \equiv [(A_1 \wedge \ldots \wedge A_n) \vee (\neg A_1 \wedge A_2 \wedge \ldots \wedge A_n \wedge p_1) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge p_1 \wedge \ldots \wedge p_n)] \wedge [A_{n+1} \vee (\neg A_n \wedge p_{n+1})]$

Après simplification on aura:
$K_\Sigma \equiv (A_1 \wedge \ldots \wedge A_n \wedge A_{n+1}) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n) \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge p_1 \wedge \ldots \wedge p_n \wedge p_{n+1})$

$K_\Sigma \equiv (A_1 \wedge \ldots \wedge A_n \wedge A_{n+1}) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n) \vee [(\neg A_1 \wedge \ldots \wedge \neg A_n \wedge p_1 \wedge \ldots \wedge p_n \wedge p_{n+1}) \wedge \mathbf{(\neg A_{n+1} \vee A_{n+1})}]$

$K_\Sigma \equiv (A_1 \wedge \ldots \wedge A_n \wedge A_{n+1}) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee \mathbf{(\neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n)} \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge \neg A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n \wedge p_{n+1}) \vee \mathbf{(\neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n \wedge p_{n+1})}$

Or $(\neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n) \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n \wedge p_{n+1}) \equiv \neg A_1 \wedge \ldots \wedge \neg A_n \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_n$.

Enfin on a:

$K_\Sigma \equiv (A_1 \wedge \ldots \wedge A_{n+1}) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_{n+1} \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_{n+1} \wedge p_1 \wedge \ldots \wedge p_{n+1})$ □

Exemple 4.4
Soit la base possibiliste : $\Sigma = \{(p, 0.9), (q, 0.7), (r, .0.5)\}$
D'après la proposition 4.4, K_Σ peut s'écrire sous la forme :
$K_\Sigma = (A_1 \wedge A_2 \wedge A_3) \vee (\neg A_1 \wedge A_2 \wedge A_3 \wedge p) \vee (\neg A_1 \wedge \neg A_2 \wedge A_3 \wedge p \wedge q) \vee (\neg A_1 \wedge \neg A_2 \wedge \neg A_3 \wedge p \wedge q \wedge r)$.

Nous nous intéressons maintenant au calcul de Comp(Σ) telle que pour toute clause p, $\Sigma \models_\pi p \Leftrightarrow$ Comp(Σ) $\models p$. En utilisant la forme de K_Σ donnée par la proposition 4.4, nous démontrons la proposition suivante :

Proposition 4.5
Soient Σ une base de croyances possibiliste et K_Σ la base de connaissances propositionnelle correspondante selon la définition 3.6. On montre alors que:
$\forall\ i \leq n : p_1 \wedge \ldots \wedge p_i \equiv ForgetVariable(\neg A_i \wedge K_\Sigma, S)$.

Preuve
$K_\Sigma \equiv$
$(A_1 \wedge \ldots \wedge A_n) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_{i-1} \wedge A_i \wedge \ldots \wedge A_n \wedge p_1 \wedge \ldots \wedge p_{i-1}) \vee$
$(\neg A_1 \wedge \ldots \wedge \neg A_{i-1} \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge p_1 \wedge \ldots \wedge p_n)$.

On voit que
$\neg A_i \wedge K_\Sigma \equiv$
$(\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n \wedge p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (\neg A_1 \wedge \ldots \wedge \neg A_n \wedge p_1 \wedge \ldots \wedge p_n)$.

$ForgetVariable(\neg A_i \wedge K_\Sigma, S) \equiv$
$(p_1 \wedge \ldots \wedge p_i \wedge ForgetVariable(\neg A_1 \wedge \ldots \wedge \neg A_i \wedge A_{i+1} \wedge \ldots \wedge A_n, S)) \vee \ldots \vee (p_1 \wedge \ldots \wedge p_n \wedge ForgetVariable(\neg A_1 \wedge \ldots \wedge \neg A_n, S))$ { car $\forall\ j = 1..n$: $\mathrm{Var}(p_j) \cap S = \varnothing$

D'autre part, $\forall\ j = 1..n$: $ForgetVariable(\neg A_1 \wedge \ldots \wedge \neg A_j \wedge A_{j+1} \wedge \ldots \wedge A_n, S) = \top$ (propriété 4 de l'oubli de variables) d'où :

$ForgetVariable(\neg A_i \wedge K_\Sigma, S) \equiv (p_1 \wedge \ldots \wedge p_i) \vee \ldots \vee (p_1 \wedge \ldots \wedge p_n) \equiv p_1 \wedge \ldots \wedge p_i$ □

Dans la proposition suivante, nous décrivons la méthode que nous proposons pour calculer Comp(Σ) en utilisant K_Σ la proposition 4.5 :

Proposition 4.6
Soient Σ une base de croyances possibiliste et K_Σ la base de connaissances propositionnelle qui lui est associée selon la définition 3.6. On a alors :

$$\text{Comp}(\Sigma) = \begin{cases} ForgetVariable(\neg A_n \wedge K_\Sigma, S) \text{ si } \Sigma \text{ est consistante.} \\ ForgetVariable(\neg A_{i-1} \wedge K_\Sigma, S) \text{ si } \Sigma \text{ est inconsistance à un degré } a_i. \end{cases}$$

Preuve
Soit Σ une base de connaissances possibiliste. Par définition de Comp(Σ), on a :
$\forall p, \Sigma \models_\pi p \Leftrightarrow$ Comp(Σ) $\models p$.

- Σ est consistante :

$\Sigma \models_\pi p \Leftrightarrow p_1 \wedge \ldots \wedge p_n \models p$
$\Leftrightarrow ForgetVariable(\neg A_n \wedge K_\Sigma, S) \models p$ {Voir Proposition 4.5.}
$\Leftrightarrow$ Comp(Σ) $\models p$

- Σ est inconsistante à un degré a_i :

$\Sigma \models_\pi p \Leftrightarrow p_1 \wedge \ldots \wedge p_{i-1} \models p$

$\Leftrightarrow$ *ForgetVariable*$(\neg A_{i-1} \wedge K_\Sigma, S) \models p$ {Voir proposition 4.5.}

$\Leftrightarrow$ Comp$(\Sigma) \models p$ □

Nous constatons que grâce à cette méthode, nous avons affaibli le problème de non décomposabilité en réduisant le nombre de variables communes entre DNNF(K_Σ) et $\neg A_i$ à A_i mais nous ne l'avons pas encore entièrement résolu : nous avons toujours Var($\neg A_i$) $\cap$ Var(DNNF(K_Σ)) $\neq \varnothing$. Ce problème peut être entièrement résolu grâce à la proposition suivante :

Proposition 4.7

Soient K une formule propositionnelle, $A \subseteq V$ un sous ensemble de variable propositionnelle et $v \in A$. CD dénote l'opération de conditionnement donnée par le définition 2.44. On montre alors que :
ForgetVariable$(\neg v \wedge K, A) \equiv$ *ForgetVariable*(CD$(K, \neg v), A - \{v\})$.

Preuve

ForgetVariable$(\neg v \wedge K, A) \equiv$ *ForgetVariable* (*ForgetVariable*$(\neg v \wedge K, v), A$- $\{v\})$

Or *ForgetVariable*$(\neg \mathrm{v} \wedge K, p) \equiv (\neg v \wedge K)_{v=\mathsf{T}} \vee (\neg v \wedge K)_{v=\bot}$

$\equiv K_{v=\bot}$

D'autre part, CD$(K, \neg v)$ revient à remplacer chaque occurrence du littéral $\neg v$ par Vrai, et chaque occurrence du littéral v par Faux. On déduit que $K_{v=\bot}$ n'est rien d'autre que CD$(K, \neg v)$.

Donc *ForgetVariable*$(\neg v \wedge K, A) \equiv$ ForgetVariable(CD$(K, \neg v), A - \{v\})$. □

L'idée alors pour calculer *ForgetVariable*$(\neg A_i \wedge$ DNNF$(K_\Sigma), S)$ est d'appliquer le conditionnement de DNNF(K_Σ) sur $\neg A_i$: une opération satisfaite par tous les langages cibles de compilation y compris DNNF si bien que CD(DNNF(K_Σ), $\neg v$) s'effectue en temps polynomial et génère un résultat DNNF. Ensuite, on oublie dans ce résultat (qui est DNNF) les variables de S-$\{A_i\}$.

4.2.4. Compilateur possibiliste DNNF

Nous pouvons à ce stade proposer un compilateur DNNF en logique possibiliste dont les différentes étape sont décrites à travers l'algorithme suivant :

Algorithme 4.3 : Compilateur possibiliste DNNF
Données : une base de croyances possibiliste $\Sigma = \{(p_i, a_i) : i = 1..n\}$
Résultat : une base propositionnelle Comp(Σ)
une base propositionnelle Comp'(Σ)
début

1 : $K_\Sigma \leftarrow$ Résultat du codage propositionnel de Σ ;
2 : DNNF(K_Σ) $\leftarrow$ Résultat de la compilation de K_Σ vers le langage DNNF ;
3 : $F \leftarrow$ *ForgetVariable*(DNNF(K_Σ), V) ;
4 : $G \leftarrow$ *ForgetLiteral*(F, *NegS*) ;
5 : Calculer *Inc*(Σ) ;
6 : si *Inc*(Σ) = 0 **alors** $v \leftarrow A_n$;
sinon si *Inc*(Σ) = a_i **alors** $v \leftarrow A_{i-1}$;
7 : $H \leftarrow$ CD(DNNF(K_Σ), $\neg v$) ;
8 : Comp(Σ) $\leftarrow$ *ForgetVariable*(H, $S - \{v\}$) ;
9 : Comp'(Σ) $\leftarrow$ DNNF(K_Σ) ;
retourner Comp(Σ), Comp'(Σ) ;

fin

Proposition 4.8
Les étapes de 3 à 8 de l'algorithme 4.3 s'effectuent en temps polynomial. De plus, Comp(Σ) est sous forme DNNF.

Preuve
L'étape 3 s'effectue en temps polynomial et le résultat $F \in$ DNNF car DNNF satisfait l'oubli de variables. L'étape 4 s'effectue également en temps polynomial car DNNF satisfait l'oubli de littéraux (proposition 4.2). L'étape 7 à son tour s'effectue en temps polynomial et H est dans DNNF car le conditionnement est satisfait par tous les langages cibles de compilation y compris DNNF.
Enfin, Il est de même pour l'étape 8, elle se fait en temps polynomial et le plus intéressant est que Comp(Σ) est dans DNNF qui permet une déduction (clausale) en temps polynomial.

Nous avons évoqué précédemment le problème de non décomposabilité qui survient au niveau de l'étape 3 de l'algorithme 4.2 associé à Comp'(Σ) pour le calcul du degré de certitude d'une formule. Il est de même, nous proposons sa résolution en utilisant les propositions 4.5 et 4.7.

Nous rappelons que p est une conséquence de Σ de degré a si et seulement si $a = Val(p, \Sigma) > Inc(\Sigma)$. Nous rappelons aussi que $Val(p, \Sigma) = Max\{a_i : \Sigma_{ai} \models p\}$. Or $\Sigma_{ai} \equiv p_1 \wedge \ldots \wedge p_i \equiv$ *ForgetVariable*($\neg A_i \wedge K_\Sigma$, S) selon la proposition 4.5. D'autre part, *ForgetVariable*($\neg A_i \wedge K_\Sigma$, S) revient à calculer *ForgetVariable*(G, S-{ A_i}) tel que $G = CD(K_\Sigma, \neg A_i)$.

En outre, et dans un but d'optimisation, nous proposons un traitement par dichotomie. L'algorithme correspondant à Comp'(Σ) se veut comme suit :

Algorithme 4.4 : Résolution de INF_DGR
Données : Comp'(Σ)
une clause p
Résultat : déterminer si $\Sigma \models (p, a_r)$
début

1 : $l \leftarrow 1$;
2 : $u \leftarrow n$;
tant que (l < u) **faire**

3 : $r \leftarrow [(l + u) / 2]$;
4 : $G \leftarrow \text{CD}(\text{C}(K_\Sigma), \neg A_r)$;
5 : $H \leftarrow ForgetVariable(G, S - \{A_r\})$;
6 : **si** $H \models p$ **alors** $u \leftarrow r$-1 ;
sinon $l \leftarrow r$;
fait
7 : si $r > Inc(\Sigma)$ alors $\Sigma \models_\pi (p, a_r)$;

fin

Proposition 4.9
L'algorithme 4.4 tel que Comp'(Σ) est issu du compilateur possibiliste DNNF décrit par l'algorithme 4.3 (Comp'(Σ) = DNNF(K_Σ)) s'effectue en temps polynomial.

Preuve
Comp'(Σ) = DNNF(K_Σ) satisfait le conditionnement, donc G est calculé en temps polynomial à l'étape 4, en plus $G \in$ DNNF.
H est calculé en temps polynomial à l'étape 5 et $H \in$ DNNF car le langage DNNF satisfait l'oubli de variables.
Enfin, DNNF satisfait aussi la déduction clausale si bien que l'étape 6 s'effectue en temps polynomial.
□

Nous illustrons le fonctionnement de notre compilateur possibiliste DNNF via l'exemple suivant :

Exemple 4.5
Soit la base possibiliste suivante :
$\Sigma = \{(\neg q \vee r, 0.8), (q, 0.6), (s, 0.6), (\neg r, 0.5), (t, 0.3)\}$

✓ **Etape 1 :**
Soient A, B, C, D quatre variables propositionnelles correspondant aux poids 0.8, 0.6, 0.5 et 0.3 respectivement. Le codage de Σ sous forme d'une base propositionnelle selon la définition 3.6 donne :
$K_\Sigma = \{\neg q \vee r \vee A, q \vee B, s \vee B, \neg r \vee C, t \vee D, \neg A \vee B, \neg B \vee C, \neg C \vee D\}$.

✓ **Etape 2 :**

On met maintenant K_Σ sous forme DNNF. Pour ce faire, on considère l'arbre de décomposition suivant selon un ordre d'élimination naturel c'est-à-dire selon l'ordre d'apparition des variables : *q*, *r*, *s*, *t*, *A*, *B*, *C*, *D*.

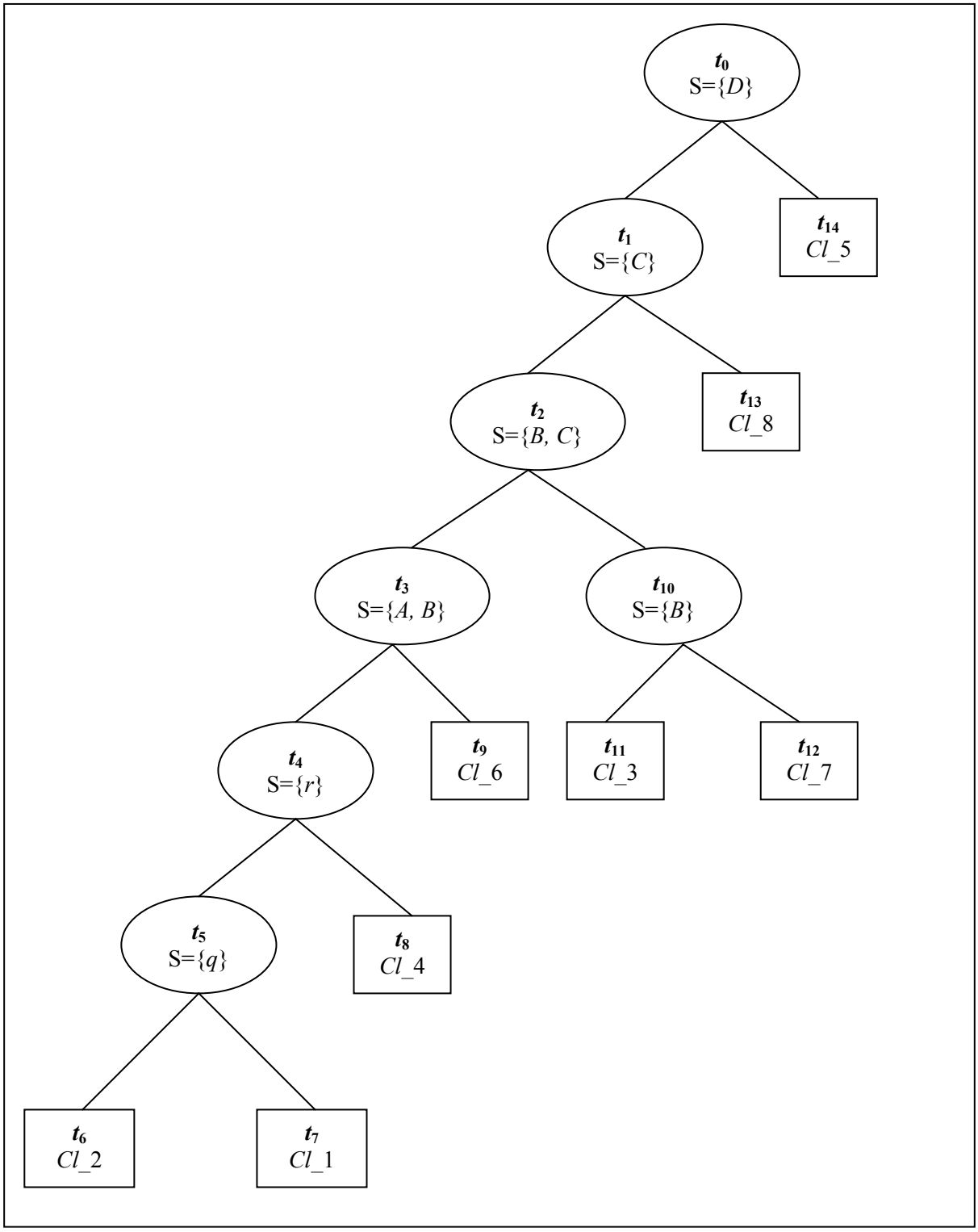


Figure 4.1. Arbre de décomposition.

On déroule ensuite la procédure dnnf donnée par la procédure 2.2 :

Initialement, on fait appel à dnnf (t0, Vrai)

dnnf (t_0, Vrai) = [$D \wedge$ dnnf(t1, D) $\wedge$ dnnf(t14, D)] $\vee$ [$\neg D \wedge$ dnnf(t1, $\neg D$) $\wedge$ dnnf(t14, $\neg D$)]
= [$D \wedge$ dnnf(t1, D)] $\vee$ [$\neg D \wedge t \wedge$ dnnf(t1, $\neg D$)]

dnnf(t1, D) = [$C \wedge$ dnnf(t2, $D \wedge C$) $\wedge$ dnnf(t13, $D \wedge C$)] $\vee$ [$\neg C \wedge$ dnnf(t2, $D \wedge \neg C$) $\wedge$ dnnf(t13, $D \wedge \neg C$)]
= [$C \wedge$ dnnf(t2, $D \wedge C$)] $\vee$ [$\neg C \wedge$ dnnf(t2, $D \wedge \neg C$)]

dnnf(t1, $\neg D$) = [$C \wedge$ dnnf(t2, $\neg D \wedge C$) $\wedge$ dnnf(t13, $\neg D \wedge C$)] $\vee$ [$\neg C \wedge$ dnnf(t2, $\neg D \wedge \neg C$) $\wedge$ dnnf(t13, $\neg D \wedge \neg C$)]
= [$\neg C \wedge$ dnnf(t2, $\neg D \wedge \neg C$)]

dnnf(t2, $D \wedge C$) = [$B \wedge$ dnnf(t3, $D \wedge C \wedge B$) $\wedge$ dnnf(t10, $D \wedge C \wedge B$)] $\vee$
[$\neg B \wedge$ dnnf(t3, $D \wedge C \wedge \neg B$) $\wedge$ dnnf(t10, $D \wedge C \wedge \neg B$)]

{dnnf(t10, $D \wedge C \wedge B$) = dnnf(t11, $D \wedge C \wedge B$) $\wedge$ dnnf(t12, $D \wedge C \wedge B$) = Vrai
dnnf(t10, $D \wedge C \wedge \neg B$) = dnnf(t11, $D \wedge C \wedge \neg B$) $\wedge$ dnnf(t12, $D \wedge C \wedge \neg B$) = s}

= [$B \wedge$ dnnf(t3, $D \wedge C \wedge B$)] $\vee$ [$\neg B \wedge s \wedge$ dnnf(t3, $D \wedge C \wedge \neg B$)]

dnnf(t2, $D \wedge \neg C$) = [$B \wedge$ dnnf(t3, $D \wedge \neg C \wedge B$) $\wedge$ dnnf(t10, $D \wedge \neg C \wedge B$)] $\vee$
[$\neg B \wedge$ dnnf(t3, $D \wedge \neg C \wedge \neg B$) $\wedge$ dnnf(t10, $D \wedge \neg C \wedge \neg B$)]
{dnnf(t10, $D \wedge \neg C \wedge B$) = dnnf(t11, $D \wedge \neg C \wedge B$) $\wedge$ dnnf(t12, $D \wedge \neg C \wedge B$) = Faux
dnnf(t10, $D \wedge \neg C \wedge \neg B$) = dnnf(t11, $D \wedge \neg C \wedge \neg B$) $\wedge$ dnnf(t12, $D \wedge \neg C \wedge \neg B$)=$s$
}
= [$\neg B \wedge s \wedge$ dnnf(t3, $D \wedge \neg C \wedge \neg B$)]

dnnf(t2, $\neg D \wedge \neg C$) = dnnf(t2, $D \wedge \neg C$) (voir cache de t2 indexé par $\neg C$)

dnnf(t3, $D \wedge C \wedge B$) = [$A \wedge$ dnnf(t4, $D \wedge C \wedge B \wedge A$) $\wedge$ dnnf(t9, $D \wedge C \wedge B \wedge A$)] $\vee$
[$\neg A \wedge$ dnnf(t4, $D \wedge C \wedge B \wedge \neg A$) $\wedge$ dnnf(t9, $D \wedge C \wedge B \wedge \neg A$)]
=[$A \wedge$ dnnf(t4, $D \wedge C \wedge B \wedge A$)] $\vee$ [$\neg A \wedge$ dnnf(t4, $D \wedge C \wedge B \wedge \neg A$)]

dnnf(t3, $D \wedge C \wedge \neg B$) = [$A \wedge$ dnnf(t4, $D \wedge C \wedge \neg B \wedge A$) $\wedge$ dnnf(t9, $D \wedge C \wedge \neg B \wedge A$)] $\vee$[$\neg A \wedge$ dnnf(t4, $D \wedge C \wedge \neg B \wedge \neg A$) $\wedge$ dnnf(t9, $D \wedge C \wedge \neg B \wedge \neg A$)]
= $\neg A \wedge$ dnnf(t4, $D \wedge C \wedge \neg B \wedge \neg A$)

dnnf(t3, $D \wedge \neg C \wedge \neg B$) = [$A \wedge$ dnnf(t4, $D \wedge \neg C \wedge \neg B \wedge A$) $\wedge$ dnnf(t9, $D \wedge \neg C \wedge \neg B \wedge A$)] $\vee$[$\neg A \wedge$ dnnf(t4, $D \wedge \neg C \wedge \neg B \wedge \neg A$) $\wedge$ dnnf(t9, $D \wedge \neg C \wedge \neg B \wedge \neg A$)]
=$\neg A \wedge$ dnnf(t4, $D \wedge \neg C \wedge \neg B \wedge \neg A$)

dnnf(t4, $D \wedge C \wedge B \wedge A$) = [$r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge A \wedge r$) $\wedge$ dnnf(t8, $D \wedge C \wedge B \wedge A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge A \wedge \neg r$) $\wedge$ dnnf(t8, $D \wedge C \wedge B \wedge A \wedge \neg r$)]
=[$r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge A \wedge \neg r$)]

dnnf(t4, $D \wedge C \wedge B \wedge \neg A$) = [$r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge \neg A \wedge r$) $\wedge$ dnnf(t8, $D \wedge C \wedge B \wedge \neg A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge \neg A \wedge \neg r$) $\wedge$ dnnf(t8, $D \wedge C \wedge B \wedge \neg A \wedge \neg r$)]
= [$r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge \neg A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge C \wedge B \wedge \neg A \wedge \neg r$)]

dnnf(t4, $D \wedge C \wedge \neg B \wedge \neg A$) = [$r \wedge$ dnnf(t5, $D \wedge C \wedge \neg B \wedge \neg A \wedge r$) $\wedge$ dnnf(t8, $D \wedge C \wedge \neg B \wedge \neg A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge C \wedge \neg B \wedge \neg A \wedge \neg r$) $\wedge$ dnnf(t8, $D \wedge C \wedge \neg B \wedge \neg A \wedge \neg r$)]
= [$r \wedge$ dnnf(t5, $D \wedge C \wedge \neg B \wedge \neg A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge C \wedge \neg B \wedge \neg A \wedge \neg r$)]

dnnf(t4, $D \wedge \neg C \wedge \neg B \wedge \neg A$) = $r \wedge$ dnnf(t5, $D \wedge \neg C \wedge \neg B \wedge \neg A \wedge r$) $\wedge$ dnnf(t8, $D \wedge \neg C \wedge \neg B \wedge \neg A \wedge r$)] $\vee$ [$\neg r \wedge$ dnnf(t5, $D \wedge \neg C \wedge \neg B \wedge \neg A \wedge \neg r$) $\wedge$ dnnf(t8, $D \wedge \neg C \wedge \neg B \wedge \neg A \wedge \neg r$)]
=$\neg r \wedge$ dnnf(t5, $D \wedge \neg C \wedge \neg B \wedge \neg A \wedge \neg r$)

dnnf(t5, $D \wedge C \wedge B \wedge A \wedge r$) = Vrai
dnnf(t5, $D \wedge C \wedge B \wedge A \wedge \neg r$)= Vrai
dnnf(t5, $D \wedge C \wedge B \wedge \neg A \wedge r$) = Vrai
dnnf(t5, $D \wedge C \wedge B \wedge \neg A \wedge \neg r$) = $\neg q$
dnnf(t5, $D \wedge C \wedge \neg B \wedge \neg A \wedge r$)= q
dnnf(t5, $D \wedge C \wedge \neg B \wedge \neg A \wedge \neg r$)=Faux
dnnf(t5, $D \wedge \neg C \wedge \neg B \wedge \neg A \wedge \neg r$)= Faux

donc :
dnnf(t4, $D \wedge C \wedge B \wedge A$) = Vrai.
dnnf(t4, $D \wedge C \wedge B \wedge \neg A$) = $r \vee (\neg r \wedge \neg q)$
dnnf(t4, $D \wedge C \wedge \neg B \wedge \neg A$) = $r \wedge q$
dnnf(t4, $D \wedge \neg C \wedge \neg B \wedge \neg A$) = Faux

dnnf(t3, $D \wedge C \wedge B$)= $A \vee [\neg A \wedge (r \vee (\neg r \wedge \neg q))]$
dnnf(t3, $D \wedge C \wedge \neg B$) = $\neg A \wedge r \wedge q$
dnnf(t3, $D \wedge \neg C \wedge \neg B$) = Faux

dnnf(t2, $D \wedge C$) = [$B \wedge$[$A \vee [\neg A \wedge (r \vee (\neg r \wedge \neg q))]$]] $\vee$ [$\neg B \wedge s \wedge \neg A \wedge r \wedge q$]
dnnf(t2, $D \wedge \neg C$) = Faux
dnnf(t2, $\neg D \wedge \neg C$) = Faux

dnnf(t1, D) = $C \wedge$[[$B \wedge$[$A \vee [\neg A \wedge (r \vee (\neg r \wedge \neg q))]$]] $\vee$ [$\neg B \wedge s \wedge \neg A \wedge r \wedge q$]]
dnnf(t1, $\neg D$) = Faux
dnnf (t_0, Vrai) = [$D \wedge$ dnnf($t1$, D)] $\vee$ [$\neg$D $\wedge$ t $\wedge$ dnnf($t1$, $\neg D$)]

DNNF(K_Σ) = $D \wedge$ $C \wedge$ [($B \wedge$ [$A \vee [\neg A \wedge (r \vee (\neg r \wedge \neg q))]$]) $\vee$ ($\neg B \wedge s \wedge \neg A \wedge r \wedge q$)]

✓ **Etape 3**

Pour calculer *ForgetVariable*(K_Σ, {q, r, s, t}), on remplace tout littéral $l = v$ ou $\neg v$ tel que $v \in$ {q, r, s, t} par Vrai :

ForgetVariable(DNNF(K_Σ), {q, r, s, t})

$= D \wedge\ C \wedge [\ (\ B \wedge [\ A \vee [\neg A \wedge (Vrai \vee (\text{Vrai} \wedge Vrai))]]\ ) \vee (\neg B \wedge \text{Vrai} \wedge \neg A \wedge \text{Vrai} \wedge \text{Vrai})\ ]$

$= D \wedge\ C \wedge [\ B \vee (\neg B \wedge \neg A)\ ]$

✓ **Etape 4**

Le calcul de *ForgetLiteral*($D \wedge\ C \wedge [\ B \vee (\neg B \wedge \neg A)\ ]$, { $\neg A$, $\neg B$, $\neg C$, $\neg D$}) se fait en remplaçant $\neg A$, $\neg B$, $\neg C$ et $\neg D$ dans $D \wedge\ C \wedge [\ B \vee (\neg B \wedge \neg A)\ ]$ par Vrai :

ForgetLiteral($D \wedge\ C \wedge [\ B \vee (\neg B \wedge \neg A)\ ]$, { $\neg A$, $\neg B$, $\neg C$, $\neg D$})

$= D \wedge\ C \wedge [\ B \vee (\text{Vrai} \wedge \text{Vrai})]$

$= D \wedge\ C$

✓ **Etape 5**

On déduit que le degré d'incohérence de Σ est $Inc(\Sigma) = 0.5$.

Si on veut utiliser la proposition 4.3, on obtient le même résultat $Inc(\Sigma) = \max(0.5, 0.3) = 0.5$

✓ **Etape 6**

v = B ;

✓ **Etape 7**

$\text{CD}(\text{DNNF}(K_\Sigma), \neg B) = D \wedge\ C \wedge s \wedge \neg A \wedge r \wedge q$

✓ **Etape 8**

$\text{ForgetVariable}(D \wedge\ C \wedge s \wedge \neg A \wedge r \wedge q, \{A, C, D\}) = s \wedge r \wedge q$

Enfin, on a :

$$\begin{cases} \text{Comp}(\Sigma) = s \wedge r \wedge q \\ \text{Comp'}(\Sigma) = D \wedge\ C \wedge [\ (\ B \wedge [\ A \vee [\neg A \wedge (r \vee (\neg r \wedge \neg q))]]\ ) \vee (\neg B \wedge s \wedge \neg A \wedge r \wedge q)\ ] \end{cases}$$

Si l'on s'intéresse au problème INF_SMPL, par exemple à partir de $\text{Comp}(\Sigma) \models r$ donc $\Sigma \models_\pi r$. Quant à l'algorithme 4.4, il nous permettra de déduire que $\Sigma \models_\pi (r, 0.6)$.

4.3. Compilateur possibiliste PI

Le langage PI des impliqués premiers a été largement étudié en Informatique et en Intelligence Artificielle en particulier. En outre, la plupart des méthodes classiques de compilation en logique propositionnelle se basent sur lui. En terme de concision, PI peut être prometteur : on sait que PI n'est pas plus concis que DNNF mais on ignore jusqu'à présent si DNNF est plus concis que PI ou non. Autrement dit, il est probable qu'il existe des formules propositionnelles n'ayant pas de formules DNNF équivalentes de taille polynomiale tandis qu'elles peuvent avoir des formules PI équivalentes de taille polynomiale. Ceci justifie notre motivation pour introduire un compilateur PI en logique possibiliste en se basant sur le compilateur possibiliste DNNF que nous avons présenté précédemment.

D'après [Darwiche and Marquis 2002], PI satisfait l'oubli de variables : les impliqués premiers de *ForgetVariable*(K, A) sont exactement les impliqués premiers de K qui ne contiennent aucune variable de A.

Nous avons un résultat similaire quant à l'oubli de littéraux : on démontre dans [Marquis 2000] que les impliqués premier de *ForgetLiteral*(K, L) sont exactement les impliqués premiers de K qui ne contiennent aucun littéral de L. Donc ces impliqués peuvent être calculés en temps polynomial en fonction de la taille de PI(K).

De plus, PI satisfait le conditionnement ainsi que la déduction clausale. Dans ce cas, le compilateur PI que nous proposons peut être directement dérivé du compilateur DNNF en compilant K_Σ vers le langage PI au lieu de le compiler vers DNNF.

Algorithme 4.5 : Compilateur possibiliste PI
Données : une base de croyances possibiliste $\Sigma = \{(p_i, a_i) : i = 1..n\}$
Résultat : une base propositionnelle Comp(Σ)
une base propositionnelle Comp'(Σ)
début

1 : $K_\Sigma \leftarrow$ Résultat du codage propositionnel de Σ ;
2 : PI(K_Σ) $\leftarrow$ Résultat de la compilation de K_Σ vers le langage PI ;
3 : $F \leftarrow$ *ForgetVariable*(PI(K_Σ), V) ;
4 : $G \leftarrow$ *ForgetLiteral*(F, *NegS*) ;
5 : Calculer *Inc*(Σ) ;
6 : si *Inc*(Σ) = 0 **alors** $v \leftarrow A_n$;
sinon si *Inc*(Σ) = a_i **alors** $v \leftarrow A_{i-1}$;
7 : $H \leftarrow$ CD(PI(K_Σ), $\neg v$) ;
8 : Comp(Σ) $\leftarrow$ *ForgetVariable*(H, $S - \{v\}$) ;
9 : Comp'(Σ) $\leftarrow$ PI(K_Σ) ;
retourner Comp(Σ), Comp'(Σ) ;

fin

Proposition 4.10
Les étapes de 3 à 8 de l'algorithme 4.5 s'effectuent en temps polynomial. En outre, Comp(Σ) appartient au langage PI.

Preuve
Cette preuve est similaire à celle de la proposition 4.8 étant donné que PI satisfait l'oubli de variables, l'oubli de littéraux et le conditionnement.

D'autre part, l'algorithme associé à Comp'(Σ) dans ce cas n'est rien d'autre que l'algorithme 4.4.

Proposition 4.11
L'algorithme 4.4 tel que Comp'(Σ) est issue du compilateur possibiliste PI décrit par l'algorithme 4.5 (Comp'(Σ) = PI(K_Σ)) s'effectue en temps polynomial.

Preuve
Preuve similaire à celle de la proposition 4.9 en substituant DNNF par PI.

4.4. *C*_Compilateur possibiliste

Nous constatons que les compilateurs possibilistes DNNF et PI que nous avons introduits ne différent qu'au niveau du langage vers lequel on compile K_Σ (soit DNNF soit PI). Nous remarquons aussi que ces compilateurs se sont avérés efficaces car les langages DNNF et PI satisfont tous les deux l'oubli de variables, l'oubli de littéraux, la déduction clausale et le conditionnement. D'autres langages peuvent être candidats pour ce faire si bien que nous proposions de définir un compilateur possibiliste plus général. Ce compilateur découle directement du compilateur DNNF (ou du compilateur PI) et se voit paramétrer par un langage cible de compilation *C* satisfaisant les propriétés précédentes.

Algorithme 4.6 : *C*_compilateur possibiliste
Données : un langage cible de compilation *C* satisfaisant certaines propriétés
une base de croyances possibiliste $\Sigma = \{(p_i, a_i) : i = 1..n\}$
Résultat : une base propositionnelle *C*_Comp(Σ)
une base propositionnelle *C*_Comp'(Σ)
début

1 : $K_\Sigma \leftarrow$ Résultat du codage propositionnel de Σ ;
2 : $C(K_\Sigma) \leftarrow$ Résultat de la compilation de K_Σ vers le langage *C* ;
3 : $F \leftarrow$ *ForgetVariable*($C(K_\Sigma)$, *V*) ;
4 : $G \leftarrow$ *ForgetLiteral*(*F*, *NegS*) ;
5 : Calculer *Inc*(Σ) ;
6 : si *Inc*(Σ) = 0 **alors** $v \leftarrow A_n$;
sinon si *Inc*(Σ) = a_i **alors** $v \leftarrow A_{i-1}$;
7 : $H \leftarrow$ CD($C(K_\Sigma)$, $\neg v$) ;
8 : *C*_Comp(Σ) $\leftarrow$ *ForgetVariable*(*H*, *S* – {*v*}) ;
9 : *C*_Comp'(Σ) $\leftarrow C(K_\Sigma)$;
retourner *C*_Comp(Σ), *C*_Comp'(Σ) ;

fin

Proposition 4.12
Soit *C* un langage de cible de compilation de connaissances propositionnelles vérifiant les propriétés suivantes :

1. *C* satisfait l'oubli de variables
2. *C* permet d'effectuer en temps polynomial l'oubli de littéraux
3. *C* satisfait le conditionnement
4. *C* satisfait la déduction clausale.

On montre alors que les étapes de 3 à 8 de l'algorithme 4.6 se font en temps polynomial et que *C*_Comp (Σ) appartient au langage *C*. De plus, l'algorithme 4.4 tel que Comp'(Σ) = *C*_Comp'(Σ) s'effectue en temps polynomial.

Preuve
Preuve similaire à celles des propositions 4.8 et 4.9.

D'après les propositions 2.2 et 2.3, aucun des langages cibles de compilation d-DNNF, sd-DNNF, FBDD, OBDD, $OBDD_<$ et IP ne peut être utilisé pour instancier *C* dans le *C*-compilateur possibiliste car aucun d'eux ne satisfait la première condition de la proposition 4.12 à savoir l'oubli de variables.

Le langage MODS par contre peut être candidat pour ce faire. En fait, MODS satisfait l'oubli de variables, le conditionnement et la déduction clausale (voir même la déduction de formule quelconque). Quant à la propriété 2, nous la démontrons à travers la proposition suivante :

Proposition 4.13
MODS permet d'effectuer l'oubli de littéraux en temps polynomial.

Preuve
On sait d'après la proposition 4.1 que la classe DNNF satisfait l'oubli de littéraux. Le langage MODS étant un sous ensemble de DNNF, donc il permet d'effectuer cette opération efficacement (on a pas besoin de prouver sa fermeture ou non par rapport à cette opération) □

Ainsi, nous avons proposé en outre un compilateur MODS en logique possibiliste. Néanmoins, d'après la proposition 2.1, nous savons que DNNF est strictement plus concis que MODS. Nous savons aussi que DNF est strictement plus concis que MODS. Nous déduisons alors qu'en terme de concision ce compilateur MODS n'est pas vraiment intéressant. Par ailleurs, MODS permet d'inférer en temps polynomial des formules propositionnelles quelconques. En d'autre terme, on est pas astreint à soumettre à la base compilée une clause comme c'est le cas pour DNNF et DNF. Le compilateur PI présente également cet avantage qui n'est pas malheureusement signifiant en le comparant au critère de concision.

Outre que son utilité en offrant un cadre unifié et précis dans l'optique d'exploiter les compilateurs propositionnels déjà existants et même ceux qui vont apparaître, le *C*_compilateur possibiliste que nous venons de décrire est qualifié d'une certaine élégance qui semble intéressante au niveau de l'implémentation voir même au niveau de l'exploitation.

4.5 Exemple d'application en sécurité

Dans le 3ème chapitre, nous avons esquissé succinctement quelques domaines d'application de la logique possibiliste. En particulier, nous avons cité le travail mené dans [Benferhat *et al*. 2003] consistant à modéliser le modèle de contrôle d'accès basé rôle RBAC (pour Role Based Access Control) en logique possibiliste afin de permettre une gestion efficace des conflits. En général, un conflit survient lorsqu'un utilisateur se trouve à la fois autorisé et interdit à exécuter une action sur un objet. L'idée sous-jacente est d'associer simplement à chaque fait de la base un degré de certitude égal

à 1. Les règles associées à l'héritage des rôles sont supposées certaines, donc auront aussi un degré égal à 1. Quant aux règles entachées d'exceptions, on considère que celles codant une situation exceptionnelle sont plus prioritaires que celles codant une situation générale. Avant de conclure ce chapitre, nous donnons un simple exemple de cette modélisation auquel nous appliquons le compilateur possibiliste DNNF que nous avons introduit.

Exemple 4.6
Soient les règles de contrôle d'accès suivantes :

« Généralement, il est interdit à un médecin d'écrire les comptes rendus des opérations. »
« Généralement, un chirurgien a la permission d'écrire les comptes rendus des opérations »
« Tout chirurgien est un médecin »
« *x* est un chirurgien. »

Nous utilisons les variables propositionnelles *h*, *m* et *r* pour coder respectivement « *x* est un chirurgien », « *x* est un médecin » et « *x* a la permission d'écrire les comptes rendus des opérations ».

Nous constatons qu'en logique propositionnelle classique, nous déduisons à la fois qu'il est permis et interdit à *x* de remplir les dossiers médicaux des opérations. Un tel conflit peut être résolu en utilisant la logique possibiliste.

La base possibiliste correspondante est la suivante : $\Sigma = \{(h, 1), (\neg h \vee m, 1), (\neg h \vee r, 0.8), (\neg m \vee \neg r, 0.3)\}$

✓ **Etape 1 :**
Soient *A*, *B*, et *C* trois variables propositionnelles correspondant aux poids 1, 0.8, et 0.3 respectivement. Le codage de Σ sous forme d'une base propositionnelle selon la définition 3.6 donne : $K_\Sigma = \{h \vee A, \neg h \vee m \vee A, \neg h \vee r \vee B, \neg m \vee \neg r \vee C, \neg A \vee B, \neg B \vee C\}$.

✓ **Etape 2 :**
On met maintenant K_Σ sous forme DNNF :
$\text{DNNF}(K_\Sigma) \equiv (\neg h \wedge A \wedge B \wedge C) \vee [h \wedge ((A \wedge B \wedge C) \vee (m \wedge \neg A \wedge C \wedge (r \vee (\neg r \wedge B))))]$

✓ **Etape 3**
Pour calculer *ForgetVariable*(K_Σ, {*h,m, r*}), on remplace tout littéral $l = v$ ou $\neg v$ tel que $v \in$ {*h, m, r*} par Vrai :
ForgetVariable(DNNF(K_Σ), {*h, m, r*})
$\equiv (\text{Vrai} \wedge A \wedge B \wedge C) \vee [\text{Vrai} \wedge ((A \wedge B \wedge C) \vee (\text{Vrai} \wedge \neg A \wedge C \wedge (\text{Vrai} \vee (\text{Vrai} \wedge B))))]$
$\equiv (A \wedge B \wedge C) \vee [(A \wedge B \wedge C) \vee (\neg A \wedge C)]$
$\equiv (A \wedge B \wedge C) \vee (\neg A \wedge C)$

✓ **Etape 4**
Le calcul de *ForgetLiteral*($(A \wedge B \wedge C) \vee (\neg A \wedge C)$, {$\neg A, \neg B, \neg C$}) se fait en remplaçant $\neg A$, $\neg B$, et $\neg C$ dans $(A \wedge B \wedge C) \vee (\neg A \wedge C)$ par Vrai :
ForgetLiteral($(A \wedge B \wedge C) \vee (\neg A \wedge C)$, {$\neg A, \neg B, \neg C$})

$\equiv (A \wedge B \wedge C) \vee (Vrai \wedge C)$
$\equiv C$

✓ **Etape 5**
On déduit que le degré d'incohérence de Σ est $Inc(\Sigma) = 0.3$.

✓ **Etape 6**
$v = B$;

✓ **Etape 7**
$CD(DNNF(K_\Sigma), \neg B) = (\neg h \wedge A \wedge Faux \wedge C) \vee [h \wedge ((A \wedge Faux \wedge C) \vee (m \wedge \neg A \wedge C \wedge (r \vee (\neg r \wedge Faux))))]$
$= h \wedge m \wedge \neg A \wedge C \wedge r$

✓ **Etape 8**
$ForgetVariable(h \wedge m \wedge \neg A \wedge C \wedge r, S-\{B\}) = h \wedge m \wedge r$

Enfin, on a : $\boldsymbol{Compiled(\Sigma) = h \wedge m \wedge r}$
A partir de *Compiled*(Σ), on déduit *r* c'est-à-dire on déduit que *x* a le droit de remplir les rapports d'opérations.

4.6. Conclusion

Dans ce chapitre, nous avons décrit notre contribution qui s'inscrit dans le cadre de la compilation de base de croyances possibilistes. En exploitant simultanément les derniers travaux menés par Marquis et Darwiche en compilation de bases de connaissances propositionnelles, et en se basant sur la méthode récemment proposée par Benferhat et Prade dédiée à la compilation d'une base de croyances possibilistes, nous avons introduit un compilateur DNNF en logique possibiliste. Nous avons en outre introduit un compilateur PI. Enfin, nous avons présenté la généralisation de ces derniers en définissant un compilateur possibiliste plus général (*C*_compilateur) dans le sens où il est paramétrable par un langage cible de compilation *C* satisfaisant certaines propriétés.

Conclusion générale

La logique possibiliste offre un outil satisfaisant en vue de représenter des connaissances incertaines, des buts avec priorité, de gérer des conflits et des exceptions. Cette logique n'est pas dénuée d'intérêt pratique. En effet, elle retrouve des applications des domaines divers tels que la sécurité et les systèmes multi-agents.

Le revers de la médaille est que le raisonnement en logique possibiliste est onéreux d'un point de vue calculatoire. Les deux relations d'inférence possibiliste, sans ou avec degré de certitude, constituent respectivement des problèmes BH_2-Complet et $\Delta_2^P[\log(n)]$.

Afin de remédier à ce problème, le paradigme de compilation de connaissances qui a fait ses preuves en logique propositionnelle classique représente une solution intéressante. Récemment, on a proposé un compilateur possibiliste DNF. Ce compilateur permet de répondre aux deux relations d'inférence possibiliste en temps polynomial par rapport à la taille de la base compilée résultante. Néanmoins, le langage DNF souffre du problème suivant : il existe des formules logiques qui n'admettent pas de représentation DNF polynomiale ce qui remet en cause l'efficacité du compilateur correspondant.

En compilation de bases de connaissances propositionnelles, on a défini d'autres langages cibles de compilation outre que DNF, et de surcroît, on les a analysé et comparé en terme de concision et d'opérations logiques qu'il permettent d'effectuer en temps polynomial.

Notre contribution s'inscrit dans le cadre de pallier la limitation du compilateur possibiliste DNF et de proposer d'autres compilateurs possibilistes plus efficaces en exploitant les autres langages cibles de compilation. Le langage DNNF suscite un intérêt particulier étant donné qu'il s'agit du langage le plus concis de tous les autres langages cibles de compilation (à part PI). Nous avons introduit dans un premier temps un compilateur possibiliste DNNF. Nous nous sommes par la suite intéressés à un langage qui a été amplement étudié en Intelligence Artificiel à savoir le langage PI (le rival de DNNF en terme de concision). En fait, on sait que PI n'est pas plus concis que DNNF mais on ignore jusqu'à présent si DNNF est plus concis que PI ou non. Nous avons également défini un compilateur possibiliste plus général qui est paramétré par un langage cible de compilation C. Outre que l'élégance offerte par ce compilateur au niveau implémentation et au niveau exploitation, ce dernier nous permet d'analyser d'une manière unifiée et précise les autres langages cibles de compilation afin de proposer d'autres compilateurs possibilistes.

Comme perspectives, nous trouvons intéressant d'exploiter la méthode de compilation propositionnelle basée impliqués premiers modulo une théorie traitable au sein du C-compilateur possibiliste. L'ensemble des clauses binaires rajoutées afin de traduire la relation de préférence entre les degrés de certitude peut déjà faire partie de la théorie traitable. Il est également pertinent d'étendre ce compilateur au cas de la logique possibiliste du premier ordre. Enfin, nous envisageons l'adapter en

vue de compiler des bases de croyances stratifiées en considérant en particulier le processus d'inférence d'ordre linéaire.

Bibliographie

[Amgoud and Prade 2003] L. Amgoud, H. Prade: A Possibilistic Logic Modeling of Autonomous Agents Negotiation. EPIA 2003: 360-365, 2003.

[Barwise 1977] J. Ed. Barwise. Handbook of mathématical logic. North-Holland, Amsterdam, 1977.

[Benferhat *et al*. 1997] S. Benferhat, D. Dubois and H. Prade. Nonmonotonic reasoning, conditional objects and possibility theory. Artificial Intelligence Journal, 92 :259-276, 1997

[Benferhat *et al*. 2003] S. Benferhat, R. El Baida, F. Cuppens: A Possibilistic Logic Encoding of Access Control. FLAIRS Conference 2003: 481-485, 2003.

[Benferhat and Prade 2005] S. Benferhat and H. Prade. Encoding formulas with partially constrained weights in possibilistic-like many-sorted propositional logic. In *Proceedings of 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 1281–1286, 2005.

[Benferhat and Parade 2006] S. Benferhat and H. Parade. Compiling possibilistic knowledge bases. A paraître dans ECAI, 2006.

[Bodlaender 1993] H. L. Bodlaender. A tourist guide through treewidth. ACTA CYBERNET-ICA 11, 1-2, 1-22, 1993.

[Boole 1854] G. Boole. Les lois de la pensée. Mathesis, 1854.

[Bryant 1986] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, *C-35*, 677–691, 1986.

[Cadoli *et al*. 1996] M. Cadoli, F. M. Donini, and M. Schaerf. Is intractability of non-monotonic reasoning a real drawback? Artificial Intelligence Journal, 88(1–2):215–251, 1996.

[Chandra and Markowsky 1978] A. K. Chandra and G. Markowsky. On the number of prime implicants. *Discrete Mathematics*, 24:7–11, 1978.

[Darwiche 1998] A. Darwiche. Model-baded diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research 8*, 165-222, 1998.

[Darwiche 2001a] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, *48* (4), 608–647, 2001.

[Darwiche 2001b] A. Darwiche. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics*, *11* (1-2), 11–34, 2001.

[Darwiche and Hopkin 2001] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143*. Springer-Verlag. 180–191, 2001.

[Darwiche and Marquis 2002] A. Darwiche and P. Marquis. A Knowledge Compilation Map. In *Journal of Artificial Intelligence Research*, Vol 17, pages 229-264, 2002.

[Darwiche 2002] A. Darwiche. A compiler for deterministic decomposable negation normal form. In *Proceedings of the18th National Conference on Artificial Intelligence (AAAI)*, pages 627–634, 2002.

[Darwiche 2004] A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 328–332, 2004.

[Davis and a*l* 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, (5)7:394–397, 1962.

[Dechter and Rish 1994] R. Dechter and H. Rish. Directional resolution: The davis-putnam procedure, revisited. In KR94, pp 134-145, 1994.

[de Kleer 1992] J. de Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (*AAAI-92*), pages 780–785, 1992.

[del Val 1994] A. del Val. Tractable databases: How to make propositional unit resolution complete through compilation. In *Proceedings of the Fourth International Conference on the Principles of Knowledge Representation and Reasoning* (*KR-94*), pages 551–561, 1994.

[Dubois *et al*. 1994] D. Dubois, J. Lang, and H. Prade. Possibilistic logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 3, pages 439–513. Oxford University Press, 1994.

[Gelfond *et al.* 1989] M. Gelfond, H. Przymusinska, and T. Przymusinsky. On the relationship between circumscription and negation as failure. In *Artificial Intelligence Journal*, 38:49–73, 1989.

[Gerey and Jonshon 1979] M. R. Garey and D. S. Johnson. *Computers and Intractability,A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, Ca, 1979.

[Gergov and Meinel 1994] J. Gergov and C. Meinel. Efficient analysis and manipulation of obdds can be extended to fbdds. *IEEE Transactions on Computers*, *43* (10), 1197–1209, 1994.

[Ghallab 1988] M.Ghallab. Compilation des connaissances. Actes des journées nationales du PRC-GRECOIA, Toulouse, 1988.

[Gogic *et al.* 1995] G. Gogic, H. Kautz, C. Papadimitriou and B.Selman. The comparative linguistics of knowledge representation. In *Proc. of the* 14*th International Joint Conference on Artificial Intelligence (IJCAI'95)*, pp. 862–869, Montreal, 1995.

[Huang and Darwiche 2004] J. Huang and A. Darwiche. Using DPLL for Efficient OBDD Construction. Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004, Revised Selected Papers, Lecture Notes in Computer Science, Volume 3542, pages 157-172, 2004.

[Huang and Darwiche 2005] J. Huang and A. Darwiche. DPLL with a Trace: From SAT to Knowledge Compilation. Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI), Edinburgh, Scotland, pages 156–162, August 2005.

[Lang 2000] J. Lang. Possibilistic logic: complexity and algorithms. In D. M. Gabbay and P. Smets, editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 5: Algorithms for Uncertainty and Defeasible Reasoning, pages 179–220. Kluwer Academic, Dordrecht, 2000.

[Lang *et al*. 2003] J. Lang, P. Liberatore, and P. Marquis. Propositional Independence - Formula-Variable Independence and Forgetting. *(Electronic) Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[Lin and Reiter 1994] F. Lin and R. Reiter. Forget it! In proc. Of the AAAI Fall Symposium on relevance, pages 154-159, New Orleans, 1994.

[Marquis 1995] P. Marquis. Knowledge compilation using theory prime implicates. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (*IJCAI-95*), pages 837–843, 1995.

[Marquis 2000] P. Marquis. Consequence finding algorithms, Vol. 5 of Handbook of Defeasible Reasoning and Uncertainty Management Systems: Algorithms for Uncertain and Defeasible Reasoning. Kluwer Academic Publishers, 2000.

[Moses and Tennenholtz 1996] Y. Moses and M. Tennenholtz. Off-line reasoning for on-line efficiency: knowledge bases. *Artificial Intelligence Journal*, 83:229–239, 1996.

[Papadimitriou 1994] C. Papadimitriou. Computational complexity. *Addison–Wesley*, 1994.

[Schrag 1996] R. Schrag. Compilation for critically constrained knowledge bases. In *Proceedings of the Thirteenth National Conferenceon Artificial Intelligence* (*AAAI-96*), pages 510–515, 1996.

[Schrag and Crawford 1996] R. Schrag and J. Crawford. Implicates and prime implicates in random 3SAT. *Artificial Intelligence Journal*, 81:199–222, 1996.

[Selman and Kautz 1991] B. Selman and H. A. Kautz. Knowledge compilation using Horn approximations. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (*AAAI-91*), pages 904–909, 1991.

[Slagle *et al.* 1970] J. R. Slagle, C. L. Chang, and R. C. T. Lee. A new algorithm for generating prime implicants. *IEEE Transactions on Computers*, 19(4):304–310, 1970.

[Tison 1967] P. Tison. Generalized consensus theory and application to the minimization of boolean functions. *IEEE transactions on electronic computers*, 4:446–456, 1967.

[Zadeh 1965] L. A. Zadeh. Fuzzy sets. Information and control, 8, 338-353, 1965.