UNIVERSITÉ D'ARTOIS

Compilation de connaissances pour le comptage de modèles

THÈSE

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Samuel Thomas

Composition du jury

Rapporteurs :	Hélène Fargier	IRIT-CNRS
	Bruno Zanuttini	Université de Caen - Basse Normandie
Examinateurs :	Frédéric Koriche	Université d'Artois (co-directeur de thèse)
	Jean-Marie Lagniez	Université d'Artois (co-encadrant)
	Pierre Marquis	Université d'Artois (co-directeur de thèse)
	Laurent Simon	Université de Bordeaux

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.fr

Mise en page avec memcril (B. Mazure, CRIL) et thloria (D. Roegel, LORIA).

À mon parrain.

Sommaire

Table des figures **vi**

Introduction générale **1**

1 Contexte de la thèse 1
2 Contributions de la thèse 2
3 Organisation du mémoire 3

Chapitre 1
Logique propositionnelle et réseaux de contraintes

1.1 La logique propositionnelle 6
1.1.1 Syntaxe 6
1.1.2 Sémantique 7
1.1.3 Formes normales 9
1.2 Les réseaux de contraintes 11
1.2.1 Syntaxe 11
1.2.2 Représentation graphique d'un réseau de contraintes 14
1.2.3 Sémantique 15
1.3 Transformation de réseaux de contraintes en formule logique 17
1.3.1 Encodage des variables 18
1.3.2 Encodage des contraintes 20

Chapitre 2
De la recherche d'une solution au comptage de modèles

2.1 Résolution du problème SAT 24
2.1.1 Solveur SAT moderne : *CDCL* 24
2.1.2 La propagation unitaire 25
2.1.3 Analyses de conflits basées sur l'analyse du graphe d'implications 27
2.1.4 Heuristiques de branchement 28

2.1.5 Redémarrages . . . 29
2.2 Résolution du problème CSP . . . 29
2.2.1 Méthode de branchement binaire . . . 30
2.2.2 Méthode de branchement non-binaire (ou n-aire) . . . 31
2.2.3 Le filtrage . . . 34
2.3 Compteurs de modèles . . . 36
2.3.1 Le comptage exact de modèles . . . 36
2.3.2 Le comptage approché de modèles . . . 38
2.3.3 Le comptage de modèles via compilation de connaissances . . . 39

Chapitre 3
La compilation de connaissances

3.1 Propriétés des langages cibles pour la compilation de formules propositionnelles . . . 44
3.1.1 Efficacité temporelle . . . 44
3.1.2 Liens entre requêtes et transformations . . . 47
3.1.3 Efficacité spatiale . . . 48
3.2 Langages cibles pour la logique propositionnelle . . . 49
3.2.1 NNF et ses sous-ensembles . . . 49
3.2.2 Les diagrammes de décision binaire . . . 53
3.2.3 Les arbres de décision (DT) . . . 57
3.2.4 Le langage AFF des formules affines . . . 60
3.3 Résultats de la carte de compilation . . . 65

Chapitre 4
Les arbres de décision affine

4.1 Les réseaux de décision affine (ADN) . . . 68
4.1.1 Les graphes de décision affine (ADG) et les arbres de décision affine étendus généralisés . . . 71
4.1.2 Le langage des arbres de décision affine étendus ($\text{EADT}_\wedge$) et ses sous-ensembles 73
4.2 Extension de la carte de compilation . . . 77
4.2.1 Les requêtes . . . 78
4.2.2 Les transformations . . . 81
4.2.3 Efficacité spatiale . . . 89
4.3 Un compilateur CNF-vers-$\text{EADT}_\wedge$ . . . 91
4.3.1 Décompositions de Shannon . . . 91
4.3.2 Algorithme de compilation . . . 93
4.3.3 Expérimentations . . . 97

4.4 Conclusion 98

Chapitre 5
La compilation de réseaux de contraintes en graphes de décision décomposables multivalués

5.1 Le langage MDDG 101
5.2 MDDG et ses voisins 105
5.3 Un compilateur CN-to-MDDG de type descendant 107
5.3.1 Les contraintes universelles 108
5.3.2 Le *caching* 109
5.3.3 L'heuristique de choix de variable 110
5.3.4 L'algorithme de compilation 111
5.4 Expérimentations 113
5.4.1 Le cadre expérimental 113
5.4.2 Les résultats obtenus 114
5.5 Conclusion 116

Conclusion générale **118**
1 Rappel des contributions 118
2 Perspectives 119

Bibliographie **120**

Table des figures

1.1 Sémantique des opérateurs logiques . . . 7
1.2 Graphe primal du graphe de contraintes de l'exemple 7. . . . 14
1.3 Graphe d'incidence du réseau de contraintes de l'exemple 7. . . . 15
1.4 Graphe de compatibilité (1.4a) et graphe d'incompatibilité (1.4b) de la contrainte C_1 du réseau de contraintes de l'exemple 7. . . . 15

2.1 Arbre binaire de recherche partiel du problème de coloriage obtenu par un algorithme à branchement binaire. . . . 31
2.2 Arbre de recherche avec branchement n-aire. . . . 32
2.3 Arbre de recherche DPLL . . . 37

3.2 Acronyme des requêtes . . . 46
3.3 Acronyme des transformations . . . 47
3.4 Une formule DNNF . . . 50
3.5 Une formule d-DNNF . . . 51
3.6 Calcul du nombre de modèles d'une formule d-DNNF uniforme . . . 53
3.7 Représentation des nœuds de décision . . . 53
3.8 Une formule Decision-DNNF . . . 54
3.9 Un vtree portant sur l'ensemble des variables $PS = \{x, y, z, u, v\}$ . . . 54
3.10 Diagramme de décision binaire ordonné représentant la formule Σ. . . . 56
3.11 Diagramme de décision binaire ordonné réduit de l'exemple 3.10 . . . 57
3.12 Diagramme de décision binaire ordonné de l'exemple 42 suivant l'ordre $x_1 <_1 \cdots <_1 x_n <_1 y_1 <_1 \cdots <_1 y_n$ . . . 58
3.13 Diagramme de décision binaire ordonné de l'exemple 42 suivant l'ordre $x_1 <_2 y_1 <_2 x_2 <_2 y_2 <_2 \cdots <_2 x_n <_2 y_n$ . . . 58
3.14 Décomposition de Shannon . . . 59
3.15 Arbre de décision équivalent à la formule Σ de l'exemple 43 . . . 60

4.1 Arbre de décision ordonné Σ''. . . . 83
4.2 Arbre de décision α_j. . . . 85
4.3 Graphe d'inclusion. $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ indique que $\mathcal{L}_1 \subseteq \mathcal{L}_2$. . . . 89
4.4 Formule résultant de l'exécution de eadt sur Σ . . . 96

5.2 Graphe d'incidence du réseau de contraintes $\mathcal{N}$ . . . 103
5.3 Graphe MDDG équivalent au réseau de contraintes de l'exemple 69. . . . 103
5.4 Un MDDG équivalent au CN de l'exemple 69. . . . 104
5.5 Requêtes. $\surd$ signifie "satisfait" et $\circ$ signifie "ne satisfait pas sauf si P = NP." . . . 106
5.6 Transformations. $\surd$ signifie "satisfait" et $\circ$ signifie "ne satisfait pas sauf si P = NP." . . . 106

5.7 Graphe de la contrainte C . . . 109
5.8 Contrainte C rendue universelle après assignation de X_2 à 1 . . . 109
5.9 Graphe primal du réseau de contraintes N_1 . . . 111
5.10 Graphe primal du réseau de contraintes N_2 . . . 111
5.11 Comparaison entre le temps de compilation en MDDG et le temps de compilation en Decision-DNNF. . . . 115
5.12 Comparaison entre la taille des formes compilées en MDDG et la taille des formes compilées en Decision-DNNF. . . . 115

Introduction générale

1 Contexte de la thèse

L'intelligence artificielle est un domaine de recherche aussi ancien que les premiers ordinateurs. Le but de l'intelligence artificielle est de permettre à un système artificiel (typiquement, un ordinateur) de résoudre des tâches dont la résolution nécessite une certaine « forme d'intelligence », comme reconnaître une phrase prononcée oralement et en comprendre le sens, raisonner à partir de connaissances, jouer à des jeux, reconnaître des objets sur une image ou encore conduire des voitures en pleine ville.

Plusieurs approches ont été développées afin de résoudre des problèmes de ce type. L'une d'elles consiste à chercher à construire un « système intelligent » qui opère sur la base de connaissances formalisées de manière déclarative. Une base de connaissances regroupe des informations de différentes natures, comme des règles, des faits, des mesures, des buts ou encore des préférences. Elles intègrent des connaissances de plus en plus volumineuses et d'une nature de plus en plus complexe.

Pour coder des connaissances afin de pouvoir raisonner et prendre des décisions à partir d'elles, un langage de représentation des connaissances est nécessaire. Le langage retenu doit être suffisamment expressif pour que les connaissances nécessaires à la résolution du problème traité puissent être effectivement représentées. En outre, il est souvent attendu que le langage retenu permette la réalisation d'algorithmes efficaces implantant les raisonnements ou les processus de décision à mettre en œuvre. Par exemple, si une application doit gérer la vente des places dans un stade de football tout en optimisant le remplissage des gradins, l'utilisateur s'attend à obtenir une réponse correcte à sa demande de places, en au plus quelques secondes. Pour la sécurité des personnes, la prise en compte de l'environnement (changeant) d'une voiture commandée par ordinateur doit pouvoir s'effectuer dans un temps très rapide.

Ainsi, divers formalismes de représentation des connaissances ont-ils été proposés et étudiés. Parmi eux figurent la logique propositionnelle (classique) et les réseaux de contraintes (booléens). Pour ces deux formalismes de représentation, un problème clé est celui de la cohérence des informations codées. Il s'agit de déterminer si la formule ou le réseau de contraintes considéré possède une solution, i.e., une affectation de valeurs aux variables qui satisfait la formule ou toutes les contraintes du réseau. Ce problème de décision est appelé problème SAT en logique propositionnelle et problème CSP pour les réseaux de contraintes.

Les traitements à réaliser pour mettre en œuvre les processus de raisonnement ou de prise de décision sont souvent très coûteux du point de vue calculatoire. Ainsi, le problème SAT fut le premier problème à avoir été montré NP-complet. À ce titre, il joue un rôle fondamental en théorie de la complexité. Grâce à l'amélioration des performances des algorithmes pour la résolution en pratique des instances SAT, la réduction vers SAT est devenue une approche classique pour la résolution de certains problèmes, tels la vérification de circuits électroniques ou la gestion de dépendances entre logiciels. En effet, bien que SAT soit un problème NP-complet, les progrès réalisés dans l'algorithmique de SAT permettent d'aborder des instances de plus en plus complexes et possédant un nombre de variables de plus en plus élevé. En pratique, les algorithmes pour SAT permettent maintenant de résoudre, après réduction à SAT, des problèmes de la vie réelle que nous pensions jusqu'alors impossible à résoudre pour un ordinateur.

Il est donc naturel de s'intéresser maintenant à d'autres problèmes, plus complexes, et d'imaginer que là aussi l'ordinateur pourra être capable d'en traiter les instances de taille raisonnable. Parmi eux figure le problème #SAT du comptage des solutions d'une formule propositionnelle ou encore celui #CSP du comptage des solutions d'un réseau de contraintes. Si déterminer l'existence d'une solution à un problème est déjà difficile du point de vue computationnel, il s'ensuit évidemment que le calcul du nombre de solutions pour ce problème est au moins aussi difficile. Le problème #SAT est connu pour être le problème #P-complet de référence (et le problème #CSP a la même complexité). Comme pour le problème SAT, différentes approches ont été proposées pour résoudre #SAT et #CSP de façon exacte ou approchée. Parmi elles, on trouve des méthodes basées sur une recherche arborescente dans l'espace des affectations de valeurs aux variables. Ces approches sont relativement proches de celles mises en œuvre pour résoudre SAT et CSP, dans le sens où la conception des algorithmes qui en relèvent pose les questions de choix de variables à affecter (donc d'heuristiques à mettre au point), de filtrages à réaliser, etc., qui se posent aussi quand il s'agit « juste » de déterminer si une solution existe. Néanmoins, les algorithmes pour #SAT et #CSP en diffèrent par le fait que la recherche ne peut pas être stoppée quand une solution a été trouvée (hormis le cas où l'on peut être sûr qu'il n'en existe pas d'autres).

2 Contributions de la thèse

Dans cette thèse, nous nous intéressons à la résolution pratique des problèmes #SAT et #CSP selon un autre type d'approche, basée sur la compilation des connaissances. Le principe est de traduire la formule propositionnelle ou le réseau de contraintes (booléen et à domaines finis) donné dans une autre représentation à partir de laquelle le comptage des solutions pourra être effectué efficacement (i.e., en temps polynomial dans la taille de la représentation produite). Cette phase de transformation, appelée compilation, n'a lieu qu'une seule fois et la représentation produite est la forme compilée de la représentation fournie en entrée. Bien entendu, on s'attend à ce que la traduction réalisée préserve les solutions. L'intérêt de l'approche par compilation réside dans le fait que la forme compilée produite va typiquement supporter aussi d'autres traitements, de façon efficace. Parmi eux on s'intéresse en particulier au conditionnement, qui est la transformation qui consiste à affecter des valeurs à certaines variables. Alors que le comptage des solutions d'une formule ou d'un réseau de contraintes conditionné est encore #P-complet, il peut être réalisé en temps polynomial lorsque la formule ou le réseau a été compilé au préalable, sous une certaine forme. Il sera ainsi possible de calculer efficacement le nombre de solutions pour différentes affectations partielles de valeurs aux variables. Un scénario applicatif où cette approche fait sens est celui de la configuration de produits combinatoires. Les affectations partielles de valeurs aux variables correspondent à des choix possibles de l'utilisateur, qui peut souhaiter connaître pour chacun d'eux le nombre de solutions compatibles. Ainsi, si la taille de la forme compilée produite reste suffisamment « petite », le temps passé à compiler pourra être compensé par le nombre de requêtes de comptages qui seront réalisées ensuite.

Si les travaux sur la compilation de connaissances en intelligence artificielle datent de la fin des années 80, l'idée de compilation est très ancienne. Par exemple, une table de logarithmes permet de retrouver facilement et rapidement le résultat d'une calcul de logarithme donné. La table n'est calculée qu'une fois pour toute, ce que nous appelons la phase hors ligne (*off-line*), puis nous n'avons plus qu'à la lire pour retrouver un résultat, ce que nous appelons la phase en ligne (*on-line*). Évidemment, plus nous souhaitons introduire de valeurs dans la table, plus celle-ci grandit et nécessite d'espace. Un compromis doit donc être trouvé entre l'espace nécessaire pour enregistrer les informations et la complexité des calculs à réaliser, reposant sur les informations stockées. Une problématique clé en compilation est de répondre à ce problème en déterminant le langage cible dans lequel la forme compilée sera représentée. Ce langage doit permettre de réaliser le plus efficacement possibles les divers traitements (requêtes et trans-

formations) demandés par l'application considérée, tout en permettant de représenter les informations de la façon la plus compacte possible.

Plus précisément, dans la suite nous introduisons et étudions de nouveaux langages cibles pour la compilation de formules propositionnelles et de réseaux de contraintes.

Dans un premier temps, nous nous sommes focalisés sur la compilation de formules de la logique propositionnelle (classique) et nous avons proposé des langages cibles pour lesquels tant le comptage de solutions que le conditionnement peuvent être réalisés en temps polynomial. Ces langages s'appuient sur le concept de nœud de décision affine, qui généralise le concept bien connu de nœud de décision, exploité dans des langages cibles existants, comme celui des arbres de décision (DT) ou encore celui des diagrammes de décision ordonnés ($\mathtt{OBDD}_<$). Nous avons ensuite développé et évalué un compilateur permettant de traduire une formule propositionnelle sous forme CNF vers ces langages, en particulier vers le langage des arbres de décision affine étendus ($\mathtt{EADT}_\wedge$). Nos résultats (tant théoriques qu'empiriques) montrent que ce langage peut conduire à des formes compilées qui sont plus succinctes que celles que l'on peut obtenir avec des compilateurs existants, comme **c2d** qui cible le langage d-DNNF [Dar01] (et plus spécifiquement, le langage Decision-DNNF [OD14]).

Dans un second temps, nous nous sommes intéressé aux langages cibles pour la compilation de réseaux de contraintes. Nous avons défini un nouveau langage cible, appelé MDDG, dont les éléments sont des graphes de décision décomposables multivalués. Il s'agit d'une extension du langage Decision-DNNF proposé pour les formules propositionnelles, au cas des réseaux de contraintes (donc, en particulier, quand les variables ne sont pas forcément booléennes). Nous avons montré qu'il est possible de compter le nombre exact de solutions d'un graphe de décision décomposable multivalué en temps polynomial mais aussi de réaliser efficacement son conditionnement. Puis, nous avons développé et évalué un compilateur permettant de transformer un réseau de contraintes en un tel graphe. Si réduire une instance de CSP à une instance de SAT pour la résoudre plus efficacement a du sens, l'exigence de préserver les solutions (qui est importante pour obtenir une forme compilée exacte) est beaucoup plus forte que celle de préserver seulement leur existence. Pour cette raison, quoiqu'il existe des transformations de réseaux de contraintes en formules propositionnelles préservant les solutions, nous avons montré expérimentalement qu'il n'est en général pas viable de les utiliser. Cela justifie l'intérêt du langage MDDG que nous avons introduit.

3 Organisation du mémoire

Le contexte de la thèse est décrit dans les chapitres 1 à 3.

Le chapitre1 présente les deux formalismes de représentation étudiés (formules propositionnelles et réseaux de contraintes). On y trouve d'abord une description de la logique propositionnelle, suivi d'une brève introduction aux approches par recherche arborescente permettant de résoudre le problème SAT. Nous définissons également les réseaux de contraintes, puis une introduction aux approches par recherche arborescente pour résoudre le problème CSP. Comme évoqué plus haut, de telles approches par recherche arborescente sont à la base de la construction de formes compilées, qui peuvent vues comme les traces du fonctionnement d'algorithmes de parcours d'arbres de recherche [HD07]. Nous présentons ensuite différentes approches pour traduire un réseau de contraintes en une formule propositionnelle. Nous décrivons enfin quelques approches développées jusqu'à présent pour compter le nombre de solutions d'une formule propositionnelle.

Le chapitre 3 est consacré à la compilation de connaissances. Ce type d'approche y est présenté et la problématique du choix d'un langage cible pour la compilation est mise en avant. Les critères de choix d'un langage cible (son efficacité temporelle, i.e., la complexité d'effectuer les requêtes et transformations d'intérêt quand l'information est représentée dans ce langage, mais aussi son efficacité

spatiale, i.e., sa capacité relative à représenter de l'information en utilisant plus ou moins d'espace) sont rappelés. Divers langages cibles existants offrant un comptage efficace des solutions (comme `d-DNNF` [Dar01, DM02] et plusieurs de ses sous-ensembles) sont définis, de façon à pouvoir les comparer aux langages qui seront introduits dans les chapitres suivants et qui forment la contribution de la thèse. À cet effet, plusieurs résultats de la carte de compilation des langages propositionnels [DM02] sont également rappelés.

Le chapitre 4 apporte les premières contribution de cette thèse. Plusieurs nouveaux langages cibles pour la compilation sont définis puis analysés selon les critères utilisés dans la carte de compilation. Un compilateur vers le langage des arbres de décision affine étendus ($\texttt{EADT}_\wedge$) est décrit, évalué et le langage $\texttt{EADT}_\wedge$ est comparé à d'autres langages cibles possibles.

Le chapitre 5 décrit le langage cible `MDDG` pour la compilation de réseau de contraintes. `MDDG` peut être vu comme la généralisation du langage `Decision-DNNF` au cas où les variables ne sont pas forcément booléennes. Ce langage est analysé et nous montrons en particulier qu'il permet de compter efficacement le nombre de solutions et de réaliser efficacement le conditionnement. Un compilateur ciblant le langage `MDDG` est décrit et évalué. Ce compilateur vise à exploiter des techniques qui n'ont pas ou ont été peu appliquées jusqu'ici pour la compilation de réseau de contraintes.

Le mémoire se termine par par une conclusion générale qui résume les résultats obtenus et dans lequel nous présentons quelques perspectives possibles pour prolonger le travail réalisé pendant cette thèse.

Chapitre 1

Logique propositionnelle et réseaux de contraintes

Sommaire

1.1 La logique propositionnelle 6
1.1.1 Syntaxe 6
1.1.2 Sémantique 7
1.1.3 Formes normales 9
1.2 Les réseaux de contraintes 11
1.2.1 Syntaxe 11
1.2.2 Représentation graphique d'un réseau de contraintes 14
1.2.3 Sémantique 15
1.3 Transformation de réseaux de contraintes en formule logique 17
1.3.1 Encodage des variables 18
1.3.2 Encodage des contraintes 20

La programmation par contraintes est un paradigme de résolution très puissant permettant de modéliser de nombreux problèmes issu de domaines variés (fouille de données, planification, ordonnancement...). Par exemple, le problème qui consiste à configurer l'assemblage d'un vélo peut être modélisé à l'aide d'un réseau de contraintes. Plus précisément, étant donné la liste des différentes parties du vélo, le cadre, les roues, le guidon, ... et une liste de contraintes (un VTT doit être monté avec de grosses roues), le but est de trouver une combinaison des éléments disponibles pour construire le vélo de manière à ce que la configuration en résultant soit cohérente au vu des spécifications (deux roues, un pédalier, deux pédales, un cadre, un guidon, deux freins, etc.) et que cette configuration satisfasse au mieux les préférences du cycliste (de grosses roues pour un VTT, un cadre léger pour un vélo de course, etc.).

Il existe plusieurs formalismes permettant de représenter un tel problème. Dans le cadre de cette thèse, nous nous sommes particulièrement intéressé à la résolution des problèmes de satisfaction de contraintes `SAT` et `CSP`. Ces derniers permettent de représenter un problème par un ensemble de variables (prenant leur valeur dans un ensemble fini de valeurs) liées par un ensemble de contraintes mathématiques ou symboliques. L'utilisation de ces paradigmes permet ainsi de séparer la partie modélisation de la partie résolution. Décorréler ces deux problématiques permet de se focaliser sur l'élaboration de méthodes efficaces pour la recherche d'une affectation qui satisfait toutes les contraintes d'un problème donné. Indiquer si une telle solution existe est appelée « problème de décision » tandis que le problème qui consiste à les compter est appelée « problème de comptage ». Ces problèmes sont connus pour être respectivement NP-complet et #P-complet.

Ainsi, l'objectif de ce chapitre est de présenter de manière formelle les problèmes de la satisfaisabilité d'une formule propositionnelle (SAT) et d'un réseau de contraintes (CSP) ainsi que quelques encodages permettant de réduire un réseau de contraintes en une formule propositionnelle.

1.1 La logique propositionnelle

Dans cette section, nous définissons la logique propositionnelle. Nous commençons par définir les éléments de la logique propositionnelle d'un point de vue syntaxique, puis du point de vue sémantique.

1.1.1 Syntaxe

Comme pour tout langage, nous avons besoin en premier lieu de définir l'alphabet de la logique propositionnelle. Autrement dit de définir les symboles pouvant être utilisés pour exprimer une formule propositionnelle.

Définition 1 (Alphabet de la logique propositionnelle) *La logique propositionnelle est constitué :*
- *d'un ensemble de variables booléennes, noté* $\mathbf{PS}$ *;*
- *des symboles* $\top$ *et* $\bot$ *représentant respectivement les constantes propositionnelles vrai et faux ;*
- *de l'ensemble des connecteurs logiques :*
  - *la négation* $\neg$ *;*
  - *le symbole de conjonction* $\wedge$ *;*
  - *le symbole de disjonction* $\vee$ *;*
  - *le symbole d'implication* $\Rightarrow$ *;*
  - *le symbole d'équivalence* $\Leftrightarrow$ *;*
  - *le symbole du ou exclusif* $\oplus$*.*
- *des symboles de ponctuation "(" et ")".*

Nous notons $\mathrm{PROP}_{\mathrm{PS}}$ l'ensemble des formules finies formées à partir de cet alphabet. Un tel langage à besoin d'une syntaxe, c'est-à-dire de règles organisant chacun de ces symboles afin de donner un sens à ces formules. Les formules formées à partir de cet alphabet suivant ces règles représentent des formules propositionnelles. Nous les appelons aussi des formules bien formées. Une formule propositionnelle est définie par induction de la manière suivante :

Définition 2 (Formule propositionnelle) *L'ensemble des formules bien formées de* $\mathrm{PROP}_{\mathrm{PS}}$ *est le plus petit ensemble tel que :*
- $\mathbf{PS} \cup \{\bot, \top\} \in \mathrm{PROP}_{\mathrm{PS}}$ *;*
- *si* $\alpha \in \mathrm{PROP}_{\mathrm{PS}}$ *, alors* $\neg\alpha \in \mathrm{PROP}_{\mathrm{PS}}$ *;*
- *si* α *et* β *sont des éléments de* $\mathrm{PROP}_{\mathrm{PS}}$*, alors* $(\alpha \wedge \beta), (\alpha \vee \beta), (\alpha \Rightarrow \beta), (\alpha \Leftrightarrow \beta)$ *et* $(\alpha \oplus \beta)$ *sont des éléments de* $\mathrm{PROP}_{\mathrm{PS}}$*.*

Remarque 1 *Dans la suite de ce manuscrit, nous choisissons de représenter toutes les formules propositionnelles par des lettres grecques majuscules (*$\Gamma, \Sigma, \ldots$*) et les variables propositionnelles par des lettres minuscules (potentiellement indicées) (*$a, b, c_1, \ldots$*).*

Exemple 1 *Soit* $\mathbf{PS} = \{a, b, c\}$*,* $\Sigma = ((a \vee b) \wedge c)$ *est une formule bien formée de* $\mathrm{PROP}_{\mathrm{PS}}$ *tandis que* $\Sigma = ((a \vee b)c)$ *n'est pas une formule bien formée.*

Soit Σ une formule propositionnelle, nous notons $Var(\Sigma)$ l'ensemble des variables propositionnelles de Σ. Un ensemble de formules $\{\Sigma_1, \Sigma_2, \ldots, \Sigma_n\}$ est dit ensemble de formules *indépendantes* si et seulement si $\forall i, j$ tel que $i \neq j$, $Var(\Sigma_i) \cap Var(\Sigma_j) = \varnothing$.

Exemple 2 *Les formules* $\Sigma = a \vee b$ *(d'où* $Var(\Sigma) = \{a, b\}$*) et* $\Sigma' = e \wedge f$ *(d'où* $Var(\Sigma') = \{e, f\}$*) sont indépendantes (car* $Var(\Sigma) \cap Var(\Sigma') = \varnothing$*) tandis que les formules* Σ *et* $\Sigma'' = a \vee e$ *(d'où* $Var(\Sigma') = \{a, e\}$*) ne sont pas indépendantes (car* $Var(\Sigma) \cap Var(\Sigma'') = \{a\}$*).*

Définition 3 (Littéral) *Un littéral positif, noté* x *est une formule propositionnelle réduite à une variable propositionnelle, et un littéral négatif, noté* $\neg x$ *est la négation de la variable propositionnelle* x*.*

Un littéral positif x est donc vrai si et seulement si la variable x est vrai. Tandis que le littéral négatif $\neg x$ est vrai si et seulement si la variable x est fausse. Ceci représente la base de la sémantique des formules propositionnelles. Dans la section suivante, nous présentons la sémantique de ses formules. C'est-à-dire les conditions sous lesquelles une formule propositionnelle est « vraie » ou « fausse ».

1.1.2 Sémantique

La sémantique de la logique propositionnelle permet de donner un sens aux formules propositionnelles. Elle permet de déterminer les conditions nécessaires pour qu'une formule soit vraie ou fausse suivant les valeurs des variables de cette formule. En effet, une formule est évaluée à *vrai* ou *faux* selon les valeurs de vérité de ses variables. La valeur de vérité d'une formule propositionnelle complexe est fonction des valeurs de vérité de ses composants. Déterminer la valeur de vérité d'une formule consiste donc à déterminer les valeurs de vérité de ses composants les plus simples, c'est-à-dire les constantes et les variables puis de simplifier la formule obtenue en utilisant les tables de vérité des connecteurs.

Comme défini précédemment, les constantes $\top$ et $\bot$ sont utilisées pour respectivement représenter les valeurs *vrai* et *faux*. Le tableau 1.1 reporte la sémantique de l'ensemble des connecteurs de la logique propositionnelle présentés dans la section précédente.

x_1	x_2	$\neg x_1$	$x_1 \wedge x_2$	$x_1 \vee x_2$	$x_1 \Rightarrow x_2$	$x_1 \Leftrightarrow x_2$	$x_1 \oplus x_2$
faux	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*	*faux*
faux	*vrai*	*vrai*	*faux*	*vrai*	*vrai*	*faux*	*vrai*
vrai	*faux*	*faux*	*faux*	*vrai*	*faux*	*faux*	*vrai*
vrai	*vrai*	*faux*	*vrai*	*vrai*	*vrai*	*vrai*	*faux*

FIGURE 1.1 – Sémantique des opérateurs logiques

Nous remarquons que la sémantique de l'opérateur d'équivalence est la négation de celle de l'opérateur du ou exclusif. De plus, la formule $x_1 \Leftrightarrow x_2$ possède la même sémantique que la formule $(x_1 \Rightarrow x_2) \wedge (x_2 \Rightarrow x_1)$.

À partir de la sémantique des opérateurs, nous sommes maintenant en mesure de définir les règles d'interprétation des formules propositionnelles à partir des valeurs données aux variables et des constantes qui les composent.

Définition 4 (Interprétation) *Une interprétation* $\mathcal{I}$ *est une application de* $\mathbf{PS}$ *dans* $\mathbb{B} = \{\bot, \top\}$ *qui attribue une valeur de vérité à chaque variable propositionnelle.*

$$\mathcal{I} : \mathbf{PS} \rightarrow \{\bot, \top\}$$

Nous notons $\mathcal{I}(x_i)$ l'interprétation de la variable x_i de V. L'interprétation $\mathcal{I}(x_i) =$ *vrai* (respectivement *faux*), noté aussi $\mathcal{I}(x_i) = \top$ (respectivement $\bot$), signifie que la valeur de la variable x_i est à *vrai* (respectivement *faux*). La valeur d'une variable peut être représentée par ses littéraux. En effet,

l'interprétation de la variable x_i, $\mathcal{I}(x_i) = \textit{vrai}$ peut être notée par le littéral positif x_i et l'interprétation $\mathcal{I}(x_i) = \textit{faux}$ peut être notée par le littéral négatif $\neg x_i$. Dans la suite de ce manuscrit nous représentons souvent une interprétation $\mathcal{I}$ par l'ensemble de ses littéraux positifs.

L'ensemble des variables sur lesquelles porte l'interprétation $\mathcal{I}$ est noté $Var(\mathcal{I})$ et son cardinal est noté $|\mathcal{I}|$. Une interprétation n'est pas forcément définie pour l'ensemble des variables propositionnelles de la formule propositionnelle Σ considérée. Dans ce cas l'interprétation, elle est dite partielle.

Définition 5 (Interprétation incomplète, partielle et complète) *Soit Σ une formule propositionnelle, une interprétation $\mathcal{I}$ construite sur un sous-ensemble des variables propositionnelles $\mathcal{V}_\mathcal{F}$ de Σ est dite :*

— *incomplète si $| \mathcal{I} | < | \mathcal{V}_\mathcal{F} |$;*
— *partielle si $| \mathcal{I} | \leq | \mathcal{V}_\mathcal{F} |$;*
— *complète si $| \mathcal{I} | = | \mathcal{V}_\mathcal{F} |$.*

Une interprétation partielle peut facilement être prolongée vers une interprétation complète pour un ensemble donné de variables de la manière suivante :

Définition 6 (Prolongement d'une interprétation) *Soient une formule propositionnelle Σ et $\mathcal{I}$ une interprétation partielle de Σ. Une interprétation complète $\mathcal{I}_c$ de Σ est un prolongement de $\mathcal{I}$ si et seulement si $Var(\Sigma) = Var(\mathcal{I}_c)$ et $\forall x_i \in Var(\mathcal{I}), \mathcal{I}(x_i) = \mathcal{I}_c(x_i)$.*

Nous pouvons maintenant définir la sémantique d'un formule propositionnelle quelconque.

Définition 7 (Sémantique d'une formule propositionnelle) *Soient ϕ, ϕ_1 et ϕ_2 des formules propositionnelles, la sémantique d'une formule Σ selon une interprétation complète $\mathcal{I}$ est définie par induction de la manière suivante :*

— $\mathcal{I}(\top) = vrai$ *et* $\mathcal{I}(\bot) = faux$ *;*
— $\mathcal{I}(\neg\phi) = vrai$ *si et seulement si* $\mathcal{I}(\phi) = faux$ *;*
— $\mathcal{I}(\phi_1 \wedge \phi_2) = vrai$ *si* $\mathcal{I}(\phi_1) = vrai$ *et* $\mathcal{I}(\phi_1) = vrai$ *;*
— $\mathcal{I}(\phi_1 \vee \phi_2) = vrai$ *si* $\mathcal{I}(\phi_1) = vrai$ *ou* $\mathcal{I}(\phi_1) = vrai$ *;*
— $\mathcal{I}(\phi_1 \Rightarrow \phi_2) = vrai$ *si* $\mathcal{I}(\phi_1) = faux$ *ou* $\mathcal{I}(\phi_1) = vrai$ *;*
— $\mathcal{I}(\phi_1 \Leftrightarrow \phi_2) = vrai$ *si* $\mathcal{I}(\phi_1) = \mathcal{I}(\phi_2)$ *;*
— $\mathcal{I}(\phi_1 \oplus \phi_2) = vrai$ *si* $\mathcal{I}(\phi_1) \neq \mathcal{I}(\phi_1)$.

Dans la suite de cette thèse, lorsqu'aucune information n'est apportée sur la nature de l'interprétation, elle est alors considérée comme complète.

Exemple 3 *Soit la formule propositionnelle $\Sigma = (x_1 \wedge \neg x_3) \vee (x_1 \oplus x_2)$ et $\mathcal{I} = \{x_1, \neg x_2, x_3\}$ est une interprétation de Σ. Nous avons donc $\mathcal{I}(x_1) = \top$, $\mathcal{I}(x_2) = \bot$ et $\mathcal{I}(x_3) = \top$. De plus, nous avons $\mathcal{I}(\Sigma) = \mathcal{I}(\top \wedge \bot) \vee \mathcal{I}(\top \oplus \bot) = \bot \vee \top = \top$.*

Une interprétation $\mathcal{I}$ *satisfait* la formule Σ si et seulement si $\mathcal{I}(\Sigma) = \top$. Dans ce cas, l'interprétation $\mathcal{I}$ est dites un *modèle* de la formule Σ, nous le notons $\mathcal{I} \models \Sigma$. Une interprétation $\mathcal{I}$ falsifie la formule Σ si $\mathcal{I}(\Sigma) = \bot$. $\mathcal{I}$ est alors appelé un contre-modèle de Σ, noté $\mathcal{I} \not\models \Sigma$. L'ensemble des modèles de la formule Σ est noté $\texttt{mod}(\Sigma)$ et le nombre de modèles est noté $\|\Sigma\|$.

Définition 8 (Formule cohérente et valide) *Soit Σ une formule propositionnelle :*

— *Σ est dite cohérente si elle admet au moins un modèle, sinon elle est dite contradictoire ;*
— *Si pour toute interprétation $\mathcal{I}$, $\mathcal{I}$ est modèle de Σ, alors Σ est dite valide.*

Nous remarquons qu'une formule Σ est cohérente si et seulement si sa négation $\neg\Sigma$ n'est pas valide.

L'ensemble des modèles d'une formule propositionnelle Σ_1 peut être inclus dans l'ensemble des modèles d'une autre formule Σ_2. Dans ce cas, la formule Σ_2 est appelée conséquence logique de Σ_1.

Définition 9 (Conséquence logique) *Si tout modèle d'une formule Σ_1 est modèle d'une formule Σ_2, noté $\Sigma_1 \models \Sigma_2$, alors Σ_2 est une conséquence logique de Σ_1.*

La conséquence logique est aussi appelée l'implication sémantique. Il existe deux formes particulières d'implication sémantique suivant des formules particulières :

Définition 10 (Impliquant / impliqué) *Soient Σ un formule, α une conjonction de littéraux et γ une disjonction de littéraux.*

- *Une conjonction de littéraux α est un **impliquant** de Σ si chaque interprétation complète $\mathcal{I}$ modèle de α est un modèle de Σ, ainsi $\alpha \models \Sigma$;*
- *Un **impliquant premier** est un impliquant minimal pour l'inclusion ;*
- *Une disjonction de littéraux γ est un **impliqué** de Σ si $\Sigma \models \gamma$;*
- *Un **impliqué premier** est un impliqué minimal pour l'inclusion.*

Deux formules propositionnelles Σ_1 et Σ_2 sont logiquement équivalentes, noté $\Sigma_1 \equiv \Sigma_2$ si et seulement si $\Sigma_1 \models \Sigma_2$ et $\Sigma_2 \models \Sigma_1$. Autrement dit, si elles ont le même ensemble de modèles $\texttt{mod}(\Sigma_1) = \texttt{mod}(\Sigma_2)$. Le théorème de la déduction permet de montrer que le problème de prouver l'implication sémantique entre deux formules revient à prouver l'incohérence d'une formule.

Définition 11 (Déduction) *Soient Σ_1 et Σ_2 deux formules propositionnelles, $\Sigma_1 \models \Sigma_2$ si et seulement si $\Sigma_1 \wedge \neg\Sigma_2$ est une formule incohérente.*

Cette propriété de déduction est très importante car elle est utilisée dans la plupart des solveurs modernes présentés dans le chapitre suivant.

Il est possible de considérer des sous-langages de la logique propositionnelle en limitant par exemple l'ensemble des connecteurs utilisés. Dans le cas où cet ensemble de connecteurs $\mathcal{C}$ permet de représenter toute formule propositionnelle Σ par une formule Σ' construite uniquement sur $\mathcal{C}$ tel que $\texttt{mod}(\Sigma) = \texttt{mod}(\Sigma')$, alors cet ensemble est dit *complet*, sinon il est dit *incomplet*.

Exemple 4 *Soit l'ensemble de connecteurs $\{\wedge, \neg\}$. Cet ensemble est complet pour la logique propositionnelle puisqu'il permet grâce aux lois de De Morgan de représenter l'ensemble des autres connecteurs logiques. La formule $x_1 \vee x_2$ par exemple peut être représenté par la formule logiquement équivalente $\neg(\neg x_1 \wedge \neg x_2)$.*

Soit l'ensemble de connecteurs $\{\wedge, \vee\}$. Cet ensemble n'est pas complet pour la logique propositionnelle puisque la formule $\neg x_4$ ne peut pas être définie grâce à ces connecteurs.

De manière analogue il est possible de contraindre la structure de la formule tout en assurant que le langage ainsi défini soit complet. Dans la suite nous présentons les différentes formes normales et nous énonçons le fait que toute formule propositionnelle peut être définie sous l'une de ces formes.

1.1.3 Formes normales

Nous nous intéressons à présent aux différentes formes normales. Plus particulièrement nous énonçons un théorème permettant de limiter l'établissement de la satisfaction d'une formule propositionnelle au cas d'une formule mise sous forme normale conjonctive.

Définition 12 (Clause et terme) *Une **clause** est une disjonction de littéraux, c'est-à-dire une formule de la forme $l_1 \vee l_2 \vee \cdots \vee l_n$. Un **terme** est une conjonction de littéraux, c'est-à-dire une formule de la forme $l_1 \wedge l_2 \wedge \cdots \wedge l_n$.*

Comme pour les interprétations, lorsqu'il n'y a aucune ambiguïté, nous représentons les clauses et les termes par leur ensemble de littéraux.

Définition 13 *Nous distinguons trois formes normales particulières pour les formules propositionnelles :*
- *Une proposition est sous forme normale négative (*`NNF` *pour « Negative Normal Form ») si elle est exclusivement constituée de conjonctions, de disjonctions et de littéraux ;*
- *Une proposition est sous forme normale disjonctive (*`DNF` *pour « Disjonctive Normal Form ») si c'est une disjonction de termes ;*
- *Une proposition est sous forme normale conjonctive (*`CNF` *pour « Conjonctive Normal Form ») si c'est une conjonction de clauses.*

Exemple 5 *Les formules $[(a \vee b) \vee ((\neg c) \wedge d)]$, $[(a \wedge b) \vee ((\neg c) \wedge d)]$ et $[(a \vee b) \wedge ((\neg c) \vee d)]$ sont respectivement sous forme normale négative, disjonctive et conjonctive.*

Remarque 2 *De la même manière que les clauses et les termes, les formules* `CNF` *et* `DNF` *sont respectivement représentées comme des ensembles de clauses et de termes.*

Comme nous pouvons le remarquer les formes normales conjonctives et disjonctives sont des cas particuliers de la forme normale négative. De plus, la propriété suivante spécifie que toute formule peut être mise sous forme normale.

Propriété 1 *Toute formule de la logique propositionnelle peut être réécrite sous toutes les formes normales de la définition 13.*

Cependant la transformation classique peut nécessiter une croissance exponentielle de la taille de l'ensemble obtenu. Néanmoins, [Tse68] propose une approche permettant de transformer en temps et espace linéaire toute formule en une formule sous forme normale conjonctive équivalente du point de vue de la satisfaisabilité. Cette transformation permet de focaliser les recherches sur le problème `SAT` au cas de formule `CNF`.

Définition 14 (Le problème `SAT`) *Le problème* `SAT` *est le problème de décision qui consiste à savoir si une formule sous forme normale conjonctive possède ou pas un modèle.*

Remarque 3 *Dans la suite de ce manuscrit nous ne faisons pas de différence entre le problème de décision* `SAT` *et le formalisme qui consiste à représenter un problème en* `CNF`*. Nous nommons cet représentation le formalisme* `SAT`*.*

Exemple 6 *Ainsi, si nous considérons le problème de configurer l'assemblage d'un vélo, il est par exemple possible de représenter le fait que l'utilisateur doive choisir parmi trois types de cadre $\{a, b, c\}$ en associant par exemple une variable booléenne x_y à chaque cadre ($y \in \{a, b, c\}$) de telle sorte qui si x_y est assignée à vraie alors le cadre y a été choisi. Afin de forcer le choix d'un unique cadre il est nécessaire d'ajouter l'ensemble de clauses $\{x_a \vee x_b \vee x_c, \neg x_a \vee \neg x_b, \neg x_a \vee \neg x_c, \neg x_b \vee \neg x_c\}$.*

La cohérence d'une formule propositionnelle est le problème de décision de référence puisque ce fut le premier prouvé NP-complet. Il représente donc le problème NP-complet de référence [Coo71]. De ce fait, l'ensemble des problèmes NP-complets se traduisent naturellement en problèmes `SAT` ou peuvent

être réduits à ce problème. Cependant comme cela a été mis en évidence dans l'exemple précédent, il est très difficile de représenter de manière naturelle un problème à l'aide du formalisme SAT. Ainsi dans la section suivante, nous présentons le langage de représentation CSP, dont le test de cohérence est aussi NP-complet, mais qui permet de modéliser plus naturellement de nombreux problèmes et qui possède une réduction polynomiale vers le formalisme SAT (voir 1.3).

1.2 Les réseaux de contraintes

Comme pour la logique propositionnelle, les réseau de contraintes modélise un problème à l'aide de contraintes portant sur une ou plusieurs variables. Cependant contrairement au formalisme SAT, le domaine des variables est multivalué. Par exemple, nous pouvons modéliser l'ensemble des cadres disponibles à l'aide d'une variable $cadre$ dont le domaine $\{a, b, c\}$. De plus, les contraintes pouvant être utilisées peuvent être mathématiques ou symboliques ($X + Y < Z$). Les contraintes lient ainsi les variables en définissant les combinaisons de valeurs acceptées des variables. Par la suite, nous nous limitons aux réseaux de contraintes *finis*, c'est-à-dire des réseaux de contraintes dans lesquels les domaines sont finis et où les valeurs des domaines sont discrets et considérés comme des entiers. Il existe d'autres types de CSP (par exemple les CSP qualitatifs) qui ne seront pas présentés dans ce manuscrit (voir [RvBW06] pour plus d'information sur le sujet).

Dans cette section, nous présentons le formalisme de représentation des connaissances par les réseaux de contraintes. Nous présentons la syntaxe et la sémantique des réseaux de contraintes.

1.2.1 Syntaxe

Un réseau de contraintes est un triplet $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, consistant en un ensemble fini de variables $\mathcal{V} = \{V_1, \ldots, V_n\}$, un ensemble de domaines $\mathcal{D} = \{D_1, \ldots, D_n\}$ et d'un ensemble fini de contraintes $\mathcal{C} = \{C_1, \ldots C_k\}$ créant des dépendances entre les variables. Chaque domaine D_i de $\mathcal{D}$ est un ensemble fini de valeurs possible pour la variable V_i. Nous notons n le nombre de variables du réseau N et d la taille du plus grand domaine.

Définition 15 (Variables et domaines) *Une variable V_i prend une valeur choisie dans un domaine D_i. Le domaine D_i est un ensemble fini de valeurs possibles pour V_i.*

Définition 16 (Contrainte) *Une contrainte C est définie par un couple $C = (S, R)$. $S = \{V_i, \ldots, V_j\}$ est un sous-ensemble de variables de l'ensemble $\mathcal{V}$ du réseau de contraintes, appelée portée (ou scope) de la contrainte. R est un prédicat sur le produit cartésien de $D_i \times \cdots \times D_j$ des domaines associés aux variables de S, appelé relation de la contrainte. La relation définit l'ensemble des instanciations autorisées pour les variables du scope.*

Pour définir les relations des contraintes de manière formelle, il est nécessaire de définir un langage. Ce langage est un langage du premier ordre dans lequel nous trouvons des fonctions (arithmétiques ($+, -, \times, /$) ou booléennes ($\vee, \wedge, \neq, \Rightarrow, \Leftrightarrow$)), des relations ($=, \neq, <, \leq, >, \geq$) et les symboles des variables ($V_1, V_i, V_8, \ldots$).

Exemple 7 *Nous définissons le réseau de contraintes $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ suivant, qui servira d'exemple support dans la suite de cette section :*

- $\mathcal{V} = \{V_1, V_2, V_3, V_4\}$;
- $\mathcal{D} = \{D_1, D_2, D_3, D_4\}$ *tels que* $D_1 = D_2 = D_3 = D_4 = \{0, 1, 2\}$;
- $\mathcal{C} = \{C_1, C_2, C_3\}$ *tels que* $C_1 = (V_1 \neq V_2), C_2 = (V_2 \neq 1) \vee (V_2 = V_3 + V_4), C_3 = (V_3 > V_4)$.

Ce problème est donc composé de quatre variables chacune ayant le même domaine de taille 3, constitué des entiers 0, 1 et 2, et il possède trois contraintes.

Nous remarquons que les contraintes d'un réseau ne portent pas toutes sur le même nombre de variables. Ce nombre appelé arité est défini comme suit :

Définition 17 (Arité) *L'arité d'une contrainte est le nombre de variables qu'elle met en jeu. Elle est donnée par le cardinal de sa portée* $\mid S \mid$*. Les contraintes dont l'arité vaut 1 sont dites unaires, celles d'arité 2 sont dites binaires et celles d'arité supérieur à 2 sont dites non binaires.*

La contrainte d'arité maximale d'un réseau de contraintes, noté r, permet de défnir la nature du réseau de contraintes. Si toutes les contraintes d'un réseau de contraintes $\mathcal{C}$ sont binaires alors $\mathcal{C}$ est dit binaire. Lorsqu'au moins une contrainte est d'arité supérieure à deux, le réseau de contraintes est dit non-binaire.

Exemple 8 *Les contraintes* C_1 *et* C_3 *de l'exemple 7 sont binaires et la contrainte* C_2 *est ternaire. L'arité maximale de* N *est* $r = 3$*, *N *est donc un réseau de contraintes non binaires.*

Remarque 4 *Un réseau de contraintes non-binaire peut toujours être transformé en un réseau de contraintes binaires en appliquant une technique de binarisation. Les deux méthodes les plus utilisées sont l'encodage du graphe dual (dual graph encoding) [DP87] et l'encodage des variables cachées (hidden variable encoding) [RPD90].*

Une relation R d'une contrainte C ayant pour portée $S = \{V_i, \ldots, V_j\}$ peut être représentée en intension ou en extension. Une relation R définie en intension est décrite avec des opérateurs mathématiques connus comme l'égalité, la différence, les relations d'ordre, la somme, etc. Elle est définie via une fonction $D_i \times \cdots \times D_j$ vers $\{0, 1\}$ supposée calculable en temps polynomial en la taille de l'entrée.

Une relation peut aussi représentée par un ensemble de n-uplets de valeurs suivant un ordre que nous considérons sur les variables de la portée de la contrainte. Dans ce cas, nous disons que la contrainte est représentée en extension. La définition d'une relation en extension est un sous-ensemble du produit cartésien des domaines des variables appartenant au *scope* de la contrainte représentant les n-uplets de valeurs autorisées (ou interdites) par la contrainte.

Définition 18 (Tuple) *Soit* $C_i = (S, R)$ *une contrainte portant sur* k *variables* $S = \{V_{i1}, V_{i1}, \ldots, V_{ik}\}$*. Un tuple* $\mathcal{T}$ *est un élément du produit cartésien de* $\prod_{j=1}^{k} D_{ij}$ *des domaines des variables de la portée de* C_i*. Un tuple qui vérifie la relation* R *de la contrainte est dit tuple support ou tuple autorisé. Un tuple qui falsifie la relation* R *de la contrainte est dit tuple conflit ou tuple interdit. L'ensemble des tuples support (respectivement conflit) est noté* $\mathcal{T}_s$ *(respectivement* $\mathcal{T}_c$*).*

Lorsqu'aucune information n'est fournie, il est admis que l'ordre utilisé sur les variables est l'ordre lexicographique. La valeur d_i affectée à la variable V_i dans $\mathcal{T}$ est notée $\mathcal{T}[V_i]$. Nous disons que la valeur d_i est assignée à la variable V_i. Une contrainte $C = (S, R)$ est satisfaite par un tuple $\mathcal{T}$ portant sur les variables S si et seulement si l'affectation des variables aux valeurs de ce tuple vérifie sa relation R.

Exemple 9 *Soient* N *le réseau de contraintes de l'exemple 7. Le tuple* $t = (2, 1, 1)$*, est un tuple support de la contrainte* C_2*. Le tuple* $t' = (1, 1, 1)$ *est un tuple conflit de la contrainte* C_2*.*

Toute contrainte décrite en intension peut être représentée en extension. En effet, il suffit simplement de dresser la liste des tuples supports de la relation. Cependant, il est important de noter que cette transformation peut produire une contrainte en extension exponentiellement plus grande que la contrainte en intension initiale. Dans le chapitre 5 nous montrons expérimentalement qu'il est préférable, dans le cadre de la compilation, de conserver la représentation initiale du problème.

Exemple 10 *Considérons la contrainte C_1 du réseau de contraintes $\mathcal{N}$ défini à l'exemple 7. Cette contrainte est représentée en intension par l'expression $V_1 \neq V_2$. Suivant l'ordre lexicographique $V_1 < V_2$, C_1 peut aussi être représentée en extension par l'ensemble de ses tuples support de la manière suivante : $\{(0,1),(0,2),(1,0),(1,2),(2,0),(2,1\}$. Elle peut aussi être représenté par l'ensemble de ses tuples conflit : $\{(0,0),(1,1),(2,2)\}$.*

Notons que lorsqu'un réseau de contraintes est entièrement défini avec des contraintes spécifiées en extension, le réseau de contraintes est dit représenté en extension. Dans le cas contraire il est dit représenté en intension.

Les contraintes globales

La notion de contrainte globale est très utilisée dans la communauté. Cependant, bien que de nombreux travaux ont été réalisés dans ce sens, il n'existe pas définition « raisonnable » de la globalité d'une contrainte. La globalité d'une contrainte est souvent comprise dans le sens de la qualité du filtrage que l'on peut effectuer avec une méthode de cohérence locale. La notion de contrainte globale a été introduite dans [Lau78]. Certaines recherches se sont naturellement tournées vers des méthodes de filtrages spécifiques à certaines contraintes. Ces recherches ont finalement donné naissance à un catalogue de contraintes pour lesquelles une méthode de filtrage dédiée est applicable [vHK06, NBR]. Le but étant d'exploiter finement la nature de la contrainte afin d'améliorer les performances des solveurs.

Ces contraintes sont dites globales car, bien qu'elles puissent être représentées par une conjonction de contraintes de manière équivalente, ne pas considérer la contrainte dans son ensemble réduit fortement le pouvoir de filtrage. Autrement dit, si le filtrage d'une contrainte est plus efficace que sur ses décompositions, alors cette contrainte est une contrainte globale. Plus généralement, nous pouvons voir une contrainte globale comme une contrainte qui capture les relations entre un nombre non prédéfini de variables. Ainsi, les contraintes globales permettent de simplifier l'expression du problème à considérer en le rendant plus lisible et en simplifiant la modélisation en grâce à l'utilisation d'un langage de plus haut niveau.

Dans le cadre de cette thèse nous nous sommes limités à trois contraintes globales particulières :

— La contrainte `allDifferent` est la première et la plus célèbre des contraintes globales. Elle fut introduite par [Lau78]. Un contrainte `allDifferent` d'arité n est définie par l'ensemble des tuples dans lesquels les n éléments ont tous une valeur différente. Un élément pouvant être soit une variable ou une constante. $\texttt{AllDifferent}(e_1,\ldots,e_n)$ est définie telle que $e_i \neq e_j$ pour tout $i \neq j$.

— La contrainte somme est une contrainte du type $\Sigma_{i=1..n}(k_i + V_i)$ op b où n est l'arité de la contrainte, k_i est un entier et V_i la i^{me} variable de la portée de la contrainte, op est un opérateur parmi les suivants $\{=,\neq,<,\leq,>,\geq\}$ et b est un entier.

— La contrainte *element* est défini par $(V_e,[e_1,e_2,\ldots,e_n],V_i)$. V_e et V_i étant des variables du réseau de contraintes et $[e_1,e_2,\ldots,e_n]$ est un tableau de n éléments. Les éléments peuvent être soit des constantes, soit des variables. Cette contrainte précise que la variable V_i doit prendre la V_e^{me} valeur du tableau en deuxième paramètre de la fonction. Par exemple, si $V_e = 2$, alors $V_i = e_2$.

Exemple 11 *Soit $N = (\mathcal{V},\mathcal{D},\mathcal{C})$ un réseau de contraintes avec $\mathcal{V} = \{V_1,V_2,V_3\}$, $D_1 = D_2 = \{1,2\}$ et $D_3 = \{1,2,3,4\}$, et trois contraintes $C_1 = (V_1 \neq V_2)$, $C_2 = (V_1 \neq V_3)$ et $C_3 = (V_2 \neq V_3)$. Un algorithme de filtrage GAC appliqué à ce réseau ne filtre aucune valeur puisqu'un support peut être trouvé pour chaque valeur de chaque variable.*

Supposons un nouveau réseau de contraintes N_2 pour lequel ces trois contraintes sont remplacées par une seule contrainte $\texttt{AllDiff}(V_1,V_2,V_3)$. Ce réseau est logiquement équivalent au premier. Cependant, les valeurs 1 et 2 sont supprimées de D_3, puisqu'elles ne possèdent pas de support.

Ainsi, considérer l'ensemble des variables dans une unique contrainte plutôt que de considérer l'ensemble des contraintes binaires formées par les couples de variables de la portée de la contrainte `allDifferent` *permet de capturer le sens de la contrainte et par conséquent de filtrer plus de valeurs.*

1.2.2 Représentation graphique d'un réseau de contraintes

Les réseaux de contraintes peuvent être associés à des structures graphiques représentant les interdépendances entre les variables ou tuples du réseau. La « macro-structure » d'un réseau de contraintes est une représentation graphique des interdépendances entre variables du réseau. Dans la suite, nous nous concentrons sur deux types de graphes décrivant la macro-structure : le graphe primal et la graphe d'incidence.

Définition 19 (Graphe primal) *Le* graphe primal *d'un réseau de contraintes $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ est donné par le graphe non-orienté $G = (\mathcal{V}, \mathcal{E})$ tel que $\{V_i, V_j\} \in \mathcal{E}$ si et seulement si il existe une contrainte $C = (S, R)$ de $\mathcal{C}$ pour laquelle $\{V_i, V_j\} \subseteq S$.*

Exemple 12 *Soit N le réseau de contraintes de l'exemple 7. La figure 1.2 représente le graphe primal de N. La variable V_3 a pour voisinage $\{V_2, V_4\}$ et son degré est égal à 2.*

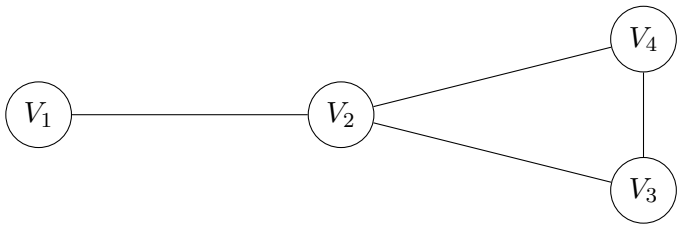


FIGURE 1.2 – Graphe primal du graphe de contraintes de l'exemple 7.

Définition 20 (Graphe d'incidence) *Le* graphe d'incidence *d'un réseau de contraintes $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ est donné par le graphe biparti $G = (\mathcal{V}, \mathcal{C}, \mathcal{E})$ où $\mathcal{V} \cup \mathcal{C}$ forme l'ensemble des sommets de G, et tel que $\{V_i, C_j\} \in \mathcal{E}$ si et seulement si V_i appartient à l'étendue de C_j.*

Exemple 13 *Considérons le réseau de contraintes N de l'exemple 7. La figure 1.3 décrit le graphe d'incidence de N où les cercles sont les variables et les rectangles sont les contraintes.*

Rapelons que la macro-structure d'un réseau de contraintes, donnée par son graphe primal ou son graphe d'incidence, ne donne aucune information sur les tuples autorisés ou interdits des contraintes du réseau. La « micro-structure » d'un réseau de contraintes permet de caratériser plus finement le réseau en représentant explicitement les interdépendances entre tuples.

Définition 21 (Micro-structure) *Etant donné un réseau de contraintes binaires $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, la* micro-structure *(ou* graphe de compatibilité*) de N est le graphe non-orienté $\mathcal{G} = (\mathcal{X}, \mathcal{E})$ tel que $\mathcal{X} = \mathcal{V} \times \mathcal{D}$ est l'ensemble des paires variable-valeur, et $\{(V_i, v_i), (V_j, v_j) \in \mathcal{E}$ si et seulement si il existe une contrainte (binaire) $C = (S, R) \in \mathcal{C}$ telle que $S = \{V_i, V_j\}$ et $(v_i, v_j) \in R$.*

De manière symmétrique, le *complément de micro-structure* (ou *graphe d'incompatibilité*) de N est le graphe $\mathcal{G} = (\mathcal{X}, \mathcal{E})$ tel que $\mathcal{X} = \mathcal{V} \times \mathcal{D}$ et $\{(V_i, v_i), (V_j, v_j) \in \mathcal{E}$ si et seulement si il existe une contrainte (binaire) $C = (S, R) \in \mathcal{C}$ telle que $S = \{V_i, V_j\}$ et $(v_i, v_j) \notin R$.

Exemple 14 *Soit N le réseau de contraintes de l'exemple 7. La figure 1.4 représente la microstructure de C_1.*

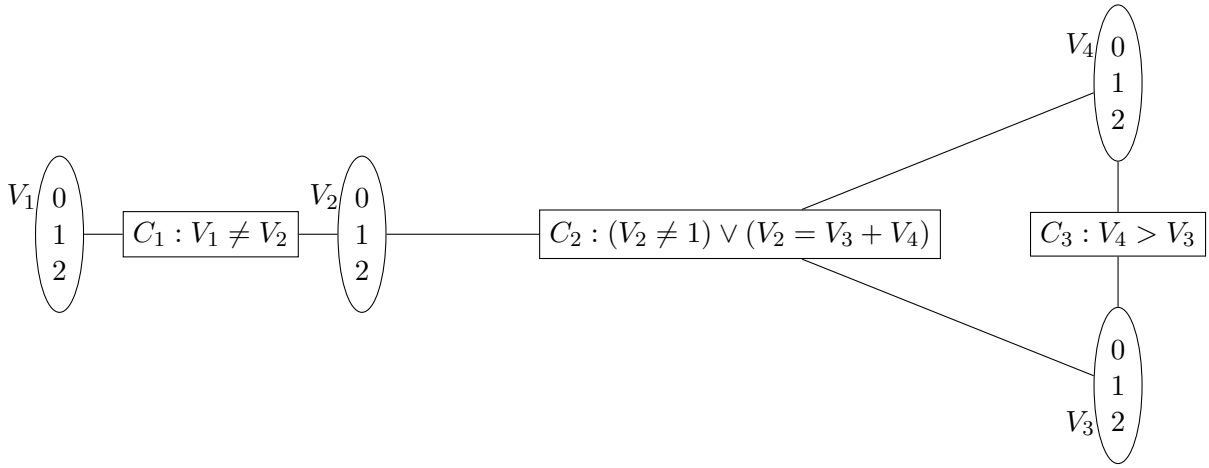


FIGURE 1.3 – Graphe d'incidence du réseau de contraintes de l'exemple 7.

(a) Microstructure support

(b) Microstructure conflit

FIGURE 1.4 – Graphe de compatibilité (1.4a) et graphe d'incompatibilité (1.4b) de la contrainte C_1 du réseau de contraintes de l'exemple 7.

1.2.3 Sémantique

Satisfaire un réseau de contraintes, c'est assigner une valeur à chaque variable de manière à ne violer aucune contrainte du problème. Une telle affection de valeurs aux variables est appelée *solution* du réseau de contrainte. Nous souhaitons déterminer si un problème formulé par un réseau de contraintes possède ou non une solution. Pour ce faire, nous définissons la sémantique des réseaux de contraintes, c'est-à-dire les conditions sous lesquelles il est satisfait. Nous devons donc définir certaines notions telles que l'affectation d'une valeur à une variable et la satisfaction d'une contrainte.

Définition 22 (Instanciation) *Soit $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes. Une instanciation $\mathcal{I}$ de $\mathcal{X} \subseteq \mathcal{V}$ est une application qui associe à chaque variable V_i une valeur $\mathcal{I}(V_i) \in D_i$. Nous notons $\mathcal{I}(\mathcal{X})$ l'ensemble de valeurs assignées par $\mathcal{I}$ aux variables de $\mathcal{X}$. $Var(\mathcal{I})$ est l'ensemble des variables instanciées par $\mathcal{I}$.*

Exemple 15 *Considérons l'exemple 7 défini précédemment. $\mathcal{I} = \{(V_1 = 1), (V_2 = 0), (V_3 = 1)\}$ est une instanciation de $\mathcal{V}$. Nous avons $\mathcal{I}(\{V_2, V_3\}) = \{(V_2 = 0), (V_3 = 1)\}$.*

Nous pouvons maintenant définir les conditions pour qu'une instanciation satisfasse une contrainte.

Définition 23 (Contrainte satisfaite ou falsifiée) *Une contrainte $C = (S, R)$ est satisfaite (respectivement falsifiée) par une instanciation telle que $S \subseteq Var(\mathcal{I})$, noté $\mathcal{I} \models C$, si et seulement si $\mathcal{I}(S) \in R$ (respectivement $\mathcal{I}(S) \notin R$).*

Exemple 16 *Considérons la contrainte $C_1 = (V_1 \neq V_2)$ de l'exemple 7. L'instanciation $\mathcal{I}_1 = \{(V_1 = 0), (V_2 = 2)\}$ satisfait la contrainte C_1. L'instanciation $\mathcal{I}_2 = \{(V_1 = 1), (V_2 = 1)\}$ falsifie C_1.*

Afin de satisfaire l'ensemble des contraintes d'un réseau de contraintes, une instanciation doit porter sur toutes les variables du réseau. Nous distinguons trois types d'instanciations.

Définition 24 (Instanciation complète, partielle et incomplète) *Soit $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes. Une instanciation $\mathcal{I}$ est dite complète, partielle ou incomplète si $\mathcal{I}$ est définie sur un ensemble de variables $\mathcal{X}$ tel que nous avons respectivement $\mathcal{X} = \mathcal{V}$, $\mathcal{X} \subseteq \mathcal{V}$ et $\mathcal{X} \subset \mathcal{V}$.*

Exemple 17 *Soit le réseau de contraintes N de l'exemple 7. Les instanciations $\mathcal{I}_1 = \{(V_2 = 2), (V_4 = 1)\}$ et $\mathcal{I}_2 = \{(V_1 = 0), (V_2 = 2), (V_3 = 0), (V_4 = 1)\}$ sont respectivement incomplète et complète. Elles sont aussi toutes les deux partielles.*

Définition 25 (Tuple support, tuple conflit) *Lorsqu'une instanciation $\mathcal{I}$ satisfait (respectivement ne satisfait pas) une contrainte C, elle est appelée tuple support (resectivement tuple conflit) de la contrainte C.*

Lorsqu'une contrainte est toujours satisfaite, quelle que soit l'instanciation considérée, cette contrainte est dite universelle. Les contraintes universelles sont des contraintes qui sont nécessairement satisfaites, quelle que soit l'affectation des variables apparaissant dans leurs portée. Nous les définissons formellement de la manière suivante :

Définition 26 (Contrainte universelle) *Soit $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes. Une contrainte $C = (S, R) \in \mathcal{C}$ est dite universelle si et seulement si C est satisfaite par toutes les instanciations de $\mathcal{V}$.*

Nous sommes maintenant en mesure de définir comment une instanciation satisfait l'ensemble des contraintes du réseau. Une telle instanciation est appelée instanciation cohérente, elle est définie de manière formelle par :

Définition 27 (Instanciation cohérente) *Soient $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes et $\mathcal{X} \subseteq \mathcal{V}$ un sous-ensemble de variables. Une instanciation $\mathcal{I}$ de $\mathcal{X}$ est cohérente si et seulement si $\forall C = (S, R) \in \mathcal{C}$, si $S \subseteq \mathcal{X}$, alors $\mathcal{I}(S)$ forme un tuple support de C.*

Lorsqu'une instanciation $\mathcal{I}$ ne satisfait pas l'une des contraintes du réseau, on dit que $\mathcal{I}$ falsifie N (noté $\mathcal{I} \not\models N$).

Exemple 18 *Soit le réseau de contraintes $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ de l'exemple 7. L'instanciation $\mathcal{I} = \{(V_1 = 2), (V_2 = 1), (V_3 = 1), (V_4 = 0)\}$ est une instanciation complète de N. $\mathcal{I}$ satisfait les trois contraintes de N, donc $\mathcal{I}$ est une solution de N.*

Un réseau de contraintes N est cohérent si et seulement si il possède au moins une solution. L'ensemble des solutions de N est noté $\texttt{mod}(N)$ et son nombre de solutions est noté $\|N\|$. Si un réseau de contraintes N possède une variable dont le domaine est vide ou si l'une de ses contraintes ne possède pas de tuple support, alors ce réseau de contraintes ne possède aucune solution. Un tel réseau de contraintes est incohérent, noté $N \equiv \bot$.

Comme nous venons de le définir, une instanciation consiste à assigner une valeur à chaque variable du réseau. Cependant, dans le cadre de la recherche de solution d'un réseau de contraintes il est possible de réfuter des valeurs du domaine d'une variable. Ces deux opérations se définissent formellement de la manière suivante :

Définition 28 (Assignation et réfutation) *Soit $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes, $V_i \in \mathcal{V}$ une variable telle que $\mid D_i \mid > 0$ et v une valeur du domaine de V_i. Une assignation $V_i = v$ (respectivement une réfutation $V_i \neq v$), noté $N_{\mid V_i = v}$ (respectivement $N_{\mid V_i \neq v}$), consiste à associer (respectivement à supprimer du domaine) la valeur v à la variable V_i. C'est-à-dire à modifier le domaine de la variable V_i tel que $D_i = \{v\}$ (respectivement $D_i = D_i \setminus \{v\}$).*

Nous remarquons qu'assigner une valeur v à une variable V consiste à réfuter l'ensemble des valeurs du domaine de V différentes de v.

Exemple 19 *Considérons l'exemple 7. L'assignation $N_{\mid V_1 = 0}$ entraîne que $D_1 = \{0\}$ et $N_{\mid V_2 \neq 0}$ entraîne que $D_2 = \{1, 2\}$.*

Il est courant lors de la construction de l'arbre de recherche par un prouveur que l'assignation et la réfutation de valeurs soient considérées. L'application d'une assignation ou d'une réfutation peut être étendue au cas d'un ensemble d'assignations et de réfutations. Un tel ensemble est appelé interprétation. Soient N un réseau de contraintes et un ensemble $\{(V_1 \bullet v_1), (V_2 \bullet v_2), \ldots (V_k \bullet v_k)\}$ tel que $(V_i \bullet v_i)$ représente une assignation ou une réfutation de V_i à v_i. Le réseau de contraintes N obtenu par application de cet ensemble est calculé inductivement par $((((N_{\mid V_1 \bullet v_1})_{\mid V_2 \bullet v_2})_{\mid \ldots})_{\mid V_k \bullet v_k})$.

1.3 Transformation de réseaux de contraintes en formule logique

Dans cette section, nous présentons quelques encodages permettant de transformer un réseau de contraintes vers la logique propositionnelle. Cette approche étant pratiquée principalement pour résoudre le problème de la cohérence, les différentes méthodes d'encodage visent à transformer les réseaux de contraintes en formules `CNF`. La transformation d'une `CNF` vers un réseau de contraintes est plutôt évidente, une `CNF` étant un réseau de contraintes dans lequel toutes les variables possèdent un domaine binaire.

Les solveurs `SAT` résolvant expérimentalement des problèmes plus difficiles, l'idée de transformer les réseaux de contraintes en `CNF` afin d'utiliser ensuite un solveur `SAT` moderne fut exploitée et l'est encore aujourd'hui.

Il existe plusieurs méthodes permettant de transformer un problème encodé en réseau de contraintes en `CNF`. Le principal objectif des solveurs étant de déterminer si l'instance donnée en entrée possède une solution, dans un souci de simplifier le problème et afin de réduire sa taille, la majorité des encodages transforment en une formule équisatisfiable. C'est-à-dire que si le problème initial possède une solution, alors la transformation de ce problème possédera une solution et si le problème initial est incohérent alors sa transformation sera aussi incohérente. Ce genre de transformation est tout à fait envisageable dans le cas où une seule solution est demandée. Cependant, comme nous le verrons dans la suite de ce manuscrit (chapitre 5), ces transformations ne sont pas utilisables dans le cas où nous souhaitons compter le nombre de modèles.

Les logiciels d'encodage tels Sugar [TTKB09, TB], Azucar [TTB12] et meSAT [SM14] utilisent des encodages performants pour les prouveurs SAT mais ne proposent pas d'encodage permettant de conserver l'équivalence. Étant donné le contexte de nos travaux (*i.e.* la compilation de connaissances) nous ne pouvons pas les utiliser. Ainsi, nous nous concentrons uniquement sur certains encodages, dans leur usage simple, en omettant volontairement des simplifications qui peuvent rendre ces encodages plus performants, mais qui feraient perdre l'équivalence logique au problème initial.

Chacun de ces encodages peut être utilisé de deux manières, en support ou en conflit. L'encodage en conflit consiste à encoder sous forme `CNF` les tuples interdit de chaque contrainte, alors que le support consiste à encoder les tuples qui la satisfont. Chacun de ces encodages est performant pour un certain

nombre de problèmes et aucun encodage spécifique n'a été démontré plus performant pour l'ensemble des problèmes. L'un des choix les plus important lorsque nous cherchons à transformer un réseau de contraintes en CNF est la représentation des domaines des variables entières par des variables booléennes auxiliaires. C'est cet encodage des domaines qui différencie principalement les différents encodages existants.

1.3.1 Encodage des variables

Il existe plusieurs façon d'encoder les variables à l'aide de variables booléenne. Dans la suite nous présentons les trois plus connus : le *direct encoding*, le *log encoding* et l'*order encoding*.

Le direct encoding Le direct encoding [dK89, Wal00] est l'encodage le plus connu et le plus utilisé lorsqu'on parle d'encodage de réseau de contraintes vers CNF. Celui-ci est le plus simple et le plus intuitif. Comme son nom l'indique il consiste à encoder directement chaque valeur du domaine des variables du réseau de contraintes.

Pour chaque variable V_i, et chaque valeur $v_{ij} \in D_i$ de son domaine, nous créons une variable booléenne x_{ij} qui a pour signification $V_i = v_{ij}$ lorsque celle-ci vaut vrai. Par exemple, si la variable V_2 du réseau de contraintes possède le domaine $D_2 = \{1, 2, 3, 4\}$, on créé les quatre variables booléennes $x_{21}, x_{22}, x_{23}, x_{24}$.

Chaque variable du réseau de contraintes doit être assignée pour trouver une solution au problème. Au moins une des variables booléennes représentant le domaine d'une variable doit donc être à vrai. Pour cela, une clause $x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}$ est créée pour chaque domaine $D_i = \{1, 2, \ldots, n\}$ des variables V_i du réseau de contraintes. Cette clause est appelée la contrainte *at least one*. Pour un réseau de contraintes $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ donné, il y a donc $\mid \mathcal{V} \mid$ clauses *at least one*, chacune d'arité $\mid D_i \mid$, la taille du domaine de la variable V_i considérée.

Dans un réseau de contraintes, une variable ne peut être assignée qu'à une et une seule valeur. Sans cette précision, notre encodage ne serait pas correct. Nous devons donc préciser que chaque variable V_i de $\mathcal{V}$ ne peut pas prendre deux valeurs différentes en même temps. Nous devons donc ajouter des contraintes indiquant qu'au plus une variable booléenne x_{ij} encodant les valeurs du domaine d'une variable V_i doit être à vrai. Pour ce faire, nous ajoutons un nombre quadratique de clauses de la forme : $\neg x_{ij'} \vee \neg x_{ij''}$ pour toutes les valeurs $j', j'' \in D_i$ telles que $j' \neq j''$. Ces clauses sont appelées les contraintes *at most one*, pour stipuler qu'au plus une valeur peut être assignée à vrai. Nous ajoutons cet ensemble une telle clause pour toutes les paires de valeur (v_{in}, v_{im}), avec $n \neq m$ du domaine de la variable. Ce qui nous donne dans le pire des cas $\frac{n \times (n-1)}{2}$ clauses supplémentaires pour encoder D_i, avec $n = \mid d_i \mid$.

Le log encoding Dans le log encoding [IM94, Wal00], le domaine d'une variable pouvant prendre d valeurs est représenté par $m = \lceil log_2 d \rceil$ variables booléennes. Chacune des 2^m combinaisons représente une assignation possible de la variable du réseau de contraintes. L'idée derrière cet encodage est d'encoder en codage binaire les valeurs du domaine des variables dans le but d'utiliser un nombre moins important de variables propositionnelles. Pour chaque variable V_i du réseau de contraintes, sont introduites les variables booléennes x_i^b avec $b = \{0, \ldots, m\}$. La variable $x_i^b = vrai$ si et seulement si, dans sa conversion en binaire, le bit b de la valeur assignés à la variable V_i est à 1.

Ceci permet à cet encodage de se passer des clauses *at least one* et des clauses *at most one*. Cependant, lorsque le cardinal du domaine n'est pas une puissance de deux, nous devons exclure les valeurs excédentaire en ajoutant des clauses pour les valeurs prohibées. Si la valeur v_h, ayant pour représentation binaire $v_{hm-1}, \ldots, v_{h0}$ (soit, $v_h = \Sigma_{b=0}^{m-1} 2^b \times v_{hb}$), n'appartient pas au domaine de la variable V_i, nous devons ajouter une contrainte stipulant que les variables représentant la valeurs de V_i ne puissent pas

représenter la valeur de v_h. Nous ajoutons donc la formule booléenne suivante : $\neg(\bigwedge_{b=0}^{m-1}(v_{hb} \Leftrightarrow x_i^b))$. Cette formule indique simplement que les bits du domaine de la variable ne peuvent pas tous avoir la même assignation que ceux d'une valeur n'appartenant pas au domaine. Cette formule peut être convertie en la clause suivante : $\bigvee_{b=0}^{m-1} v_{hb} \oplus x_i^b$. En effet, effectuer la négation d'une équivalence logique entre deux formules reviens à faire un ou exclusif entre ces deux formules. La variable v_{hb} étant une valeur définie, cela donne bien une clause de taille m définie sur les variables x_i^b. L'encodage des domaines des variables ne contient donc qu'une liste des valeurs prohibées pour chaque variable du réseau de contraintes.

Exemple 20 *Soit N_t le réseau de contraintes de l'exemple 7. L'encodage des variables se fait de la manière suivante. Les variables possèdent tous un domaine avec trois valeurs, elles sont donc chacune représentées par $\lceil log_2(3) \rceil = 2$ variables booléennes : v_{11}, v_{10}, v_{21}, v_{20}, v_{31}, v_{30} et v_{41}, v_{40}.*

La taille des domaines des variables n'étant pas une puissance de deux, nous devons prohiber la valeur 4 dans leur représentation en clause. Par exemple, pour la variable V_1 la valeur 4 est prohibée par la clause $\neg v_{11} \vee \neg v_{10}$.

Nous remarquons via l'exemple 20 que le nombre de clauses générées pour l'encodage des domaines des variables du réseau de contraintes est fortement réduit. Cependant, comme nous le verrons par la suite les clauses générées afin d'encoder les contraintes sont plus grandes. Nous verrons dans le chapitre 5 que, dans le cadre de la compilation, le fait d'avoir de grande clauses peut être problématique.

Le order encoding L' *order encoding* [BB03, TTKB09], aussi connu sous le nom de *sequential counter encoding*, consiste en une représentation particulière du domaine des variables du réseau de contraintes. En effet, pour chaque variable V_i de domaine $D_i = \{v_{i1}, \ldots, v_{in}\}$, nous créons un vecteur de taille $n-1$ de variables $[x_1^i, \ldots, x_n^i - 1]$. Chacune de ces variables x_j^i signifie que $V_i > v_j$ si elle est à vrai et $V_i \leq v_j$ sinon. Pour stipuler l'affectation de la variable $V_i = v_j$, les premières $j-1$ variables sont assignées à $vrai$ et les autres sont assignées à $faux$. Par exemple, soit la variable V_i de domaine $D_i = \{1, 2, 3, 4, 5\}$, les affectations $V_i = 1$, $V_i = 3$ et $V_i = 5$ sont respectivement représentées par $[0, 0, 0, 0]$, $[1, 1, 0, 0]$ et $[1, 1, 1, 1]$.

Un tel encodage nécessite deux propriétés :

— si $x_i^v = vrai$, alors $x_j^v = vrai$ pour tout j tel que $1 \leq j \leq i$;

— Si $x_i^v = faux$, alors $x_j^v = faux$ pour tout j tel que $i \leq j \leq n-1$.

Afin de satisfaire cette propriété, nous devons créé l'ensemble de clauses suivante :

$$\bigwedge_{i=1}^{n-2} \neg(\neg x_i^v \wedge x_{i+1}^v) \equiv \bigwedge_{i=1}^{n-2}(x_i^v \vee \neg x_{i+1}^v)$$

L'assignation d'une variable V_i du réseau de contrainte à une valeur $V_i = v_j$ est représentée en fixant la valeur des variables $x_{j-1}^{V_i} = vrai$ et $x_j^{V_i} = faux$. $V_i \neq v_j$ est représenté en ajoutant $x_{j-1}^{V_i} = x_j^{V_i}$. Pour la plus petite borne du domaine, nous supposons qu'il y a une variable implicite que l'on ne spécifie pas, puisqu'elle serait toujours à $vrai$. Cela permet de réduire le nombre de variables propositionnelles nécessaire pour représenter le réseau de contraintes.

Le principal avantage de cet encodage est qu'il permet la représentation d'intervalles de domaines et la propagation de leurs bornes. Il est en effet possible de restreindre le domaine d'une variable entre les bornes $i, \ldots, j$ en imposant $x_{i-1}^v = vrai$ et $x_j^v = faux$.

Bien qu'étant un des encodages les plus performants sur des problèmes utilisant des contraintes arithmétique, l'*order encoding* se montre moins performant sur certains types de problèmes ou de contraintes. Par exemple, il a été montré que le *direct encoding* est plus performant que l'*order encoding* lorsque le réseau de contraintes est principalement constitué de contraintes `allDifferent`.

1.3.2 Encodage des contraintes

Dans la suite suite nous présentons deux méthodes d'encodages des contraintes. Nous discutons aussi de certaines améliorations pouvant être réalisées en fonction de l'encodage des variables utilisé.

Le conflict encoding Le principe du *conflict encoding* est d'encoder les contraintes par la liste des leurs tuples interdits. Pour cela, pour chaque tuple interdit, nous créons une clause, appelée clause conflit, représentant ce tuple. Pour une contrainte dont la portée est $S = \{V_i, \ldots, V_j\}$ pour chaque tuple conflit $V_i \neq v, \ldots, V_j \neq v'$, cette liste de valeurs interdites pour ces variables est représentée par la clause conflit de la forme $\neg x_{iv} \vee \cdots \vee \neg x_{jv'}$, où $\neg x_{iv}$ représente $V_i \neq v$.

Exemple 21 *Considérons le réseau de contraintes de l'exemple 7. Supposons que le codage des variables a été réalisé à l'aide du direct encoding. Nous avons pour l'encodage de ces dernières les clauses suivantes :*

$$\begin{array}{llll} v_{10} \vee v_{11} \vee v_{12} & \neg v_{10} \vee \neg v_{11} & \neg v_{10} \vee \neg v_{12} & \neg v_{11} \vee \neg v_{12} \\ v_{20} \vee v_{21} \vee v_{22} & \neg v_{20} \vee \neg v_{21} & \neg v_{20} \vee \neg v_{22} & \neg v_{21} \vee \neg v_{22} \\ v_{30} \vee v_{31} \vee v_{32} & \neg v_{30} \vee \neg v_{31} & \neg v_{30} \vee \neg v_{32} & \neg v_{31} \vee \neg v_{32} \\ v_{40} \vee v_{41} \vee v_{42} & \neg v_{40} \vee \neg v_{41} & \neg v_{40} \vee \neg v_{42} & \neg v_{41} \vee \neg v_{42} \end{array}$$

L'ensemble des clauses permettant de coder la contrainte C_1 sont :

$$\neg v_{10} \vee \neg v_{20} \qquad \neg v_{11} \vee \neg v_{21} \qquad \neg v_{12} \vee \neg v_{22}$$

L'ensemble des clauses permettant de coder la contrainte C_2 sont :

$$\begin{array}{lll} \neg v_{21} \vee \neg v_{30} \vee \neg v_{40} & \neg v_{21} \vee \neg v_{30} \vee \neg v_{42} & \neg v_{21} \vee \neg v_{31} \vee \neg v_{41} \\ \neg v_{21} \vee \neg v_{31} \vee \neg v_{42} & \neg v_{21} \vee \neg v_{32} \vee \neg v_{40} & \neg v_{21} \vee \neg v_{32} \vee \neg v_{41} \\ \neg v_{21} \vee \neg v_{32} \vee \neg v_{41} & & \end{array}$$

L'ensemble des clauses permettant de coder la contrainte C_3 sont :

$$\begin{array}{lll} \neg v_{40} \vee \neg v_{30} & \neg v_{40} \vee \neg v_{31} & \neg v_{40} \vee \neg v_{32} \\ \neg v_{41} \vee \neg v_{31} & \neg v_{41} \vee \neg v_{32} & \neg v_{42} \vee \neg v_{32} \end{array}$$

La conjonction de ces clauses représente bien une formule `CNF` *logiquement équivalente au réseau de contrainte N encodée via le conflict encoding.*

Le support encoding Le support encoding [Gen02] consiste à encoder les tuples supports de chaque contrainte en un ensemble de clauses support. Pour chaque tuple d'une contrainte, où la variable V_i est affecté à la valeur v_{i_1}, nous ajoutons une clause indiquant les combinaisons de valeurs que peuvent prendre les autres variables dans le *scope* de la contrainte. Si la variable $V_i = v_{i_1}$ apparaît dans les tuples support avec la variable V_j avec pour valeur $V_j = v_{j_1}, V_j = v_{j_2}, \ldots, V_j = V_{j_k}$, alors nous imposons que $x_{ii_1} \rightarrow x_{jj_1} \vee x_{jj_2} \vee \cdots \vee x_{jj_k}$. Nous ajoutons donc la clause $\neg x_{ii_1} \vee x_{jj_1} \vee x_{jj_2} \vee \cdots \vee x_{jj_k}$. Si une affectation $V_i = v_{i_1}$ ne supporte aucune valeur, la clause support sera réduite à la clause unitaire $\neg x_{ii_1}$. Cela indique que l'on interdit la valeur v_{i_1} pour la variable V_i. Si au contrainte l'affectation $V_i = v_{i_1}$ supporte toutes les valeurs d'une variable V_j, aucune clause n'est requise. En effet, ajouter une clause permettant toutes les valeurs pour la variable V_j n'ajoute aucune contrainte sur le domaine de celle-ci.

Exemple 22 *Considérons une nouvelle fois le réseau de contraintes de l'exemple 7. Supposons que le codage des variables a aussi été réalisé à l'aide du direct encoding.*

L'ensemble des clauses permettant de coder la contrainte C_1 sont :

$$\begin{array}{lll} \neg v_{10} \vee v_{21} \vee v_{22} & \neg v_{11} \vee v_{20} \vee v_{22} & \neg v_{12} \vee v_{20} \vee v_{21} \\ \neg v_{20} \vee v_{11} \vee v_{12} & \neg v_{21} \vee v_{10} \vee v_{12} & \neg v_{22} \vee v_{10} \vee v_{11} \end{array}$$

L'ensemble des clauses permettant de coder la contrainte C_2 sont :

$$\begin{array}{lll} \neg v_{20} \vee v_{30} \vee v_{31} \vee v_{32} & \neg v_{20} \vee v_{40} \vee v_{41} \vee v_{42} & \neg v_{21} \vee v_{30} \vee v_{31} \\ \neg v_{21} \vee v_{40} \vee v_{41} & \neg v_{22} \vee v_{30} \vee v_{31} \vee v_{32} & \neg v_{22} \vee v_{40} \vee v_{41} \vee v_{42} \\ \neg v_{30} \vee v_{20} \vee v_{21} \vee v_{22} & \neg v_{30} \vee v_{40} \vee v_{41} \vee v_{42} & \neg v_{31} \vee v_{20} \vee v_{21} \vee v_{22} \\ \neg v_{31} \vee v_{40} \vee v_{41} \vee v_{42} & \neg v_{32} \vee v_{20} \vee v_{21} \vee v_{22} & \neg v_{32} \vee v_{40} \vee v_{41} \vee v_{42} \\ \neg v_{40} \vee v_{20} \vee v_{21} \vee v_{22} & \neg v_{40} \vee v_{30} \vee v_{31} \vee v_{32} & \neg v_{41} \vee v_{20} \vee v_{21} \vee v_{22} \\ \neg v_{41} \vee v_{30} \vee v_{31} \vee v_{32} & \neg v_{42} \vee v_{20} \vee v_{21} \vee v_{22} & \neg v_{42} \vee v_{30} \vee v_{31} \vee v_{32} \end{array}$$

L'ensemble des clauses permettant de coder la contrainte C_3 sont :

$$\begin{array}{lll} \neg v_{40} & \neg v_{41} \vee v_{30} & \neg v_{42} \vee v_{30} \vee v_{31} \\ \neg v_{32} & \neg v_{31} \vee v_{42} & \neg v_{30} \vee v_{41} \vee v_{42} \end{array}$$

La conjonction de ces clauses et de celles permettant de modéliser les domaines des variables est bien une formule `CNF` *équivalente pour la satisfiabilité au réseau de contrainte N encodée via le support encoding.*

L'exemple 7 est représentatif de comment le *support encoding* se comporte en pratique. En effet, dans la plupart des situations cet encodage nécessite en moyenne bien plus de clauses. Cependant, il a été prouvé que pour le *support encoding*, la propagation unitaire était équivalent à l'arc-cohérence du réseau de contraintes initial [Gen02, PJ10]. Il a été montré que dans le cas du *conflict encoding* il est nécessaire d'utiliser un filtrage plus coûteux que la propagation unitaire (l'*hyper binary resolution*) afin d'obtenir le même comportant [DFW02]. Plus précisément, il a aussi été montré que les clauses supports peuvent être déduites depuis le *conflict encoding* en appliquant plusieurs étapes de cette étape de filtage.

Encodage mixte Comme nous pouvons l'observer sur les deux derniers exemples, le nombre de clauses nécessaires à la représentation d'une contrainte peut varier de manière significative en fonction de l'encodage utilisé. Dans le cadre de la compilation de connaissances la taille de l'entrée peut s'avérer importante quant au performance du compilateur. En effet, la procédure de *caching* dépend fortement de cette taille (voir le chapitre 2 suivant pour plus d'informations à ce sujet). Il peut donc être intéressant de mixer les deux encodages en fonction du comportement de ces derniers. Plus précisément, pour une contrainte C_i, si le *support encoding* nécessite moins de clauses que le *conflict encoding* pour représenter C_i alors *support encoding* est utilisé, sinon le *conflict encoding* est utilisé pour représenter C_i.

Exemple 23 *Considérons une nouvelle fois le réseau de contraintes de l'exemple 7. Supposons que le codage des variables a aussi été réalisé à l'aide du direct encoding.*

L'ensemble des clauses permettant de coder la contrainte C_1 sont :

$$\neg v_{10} \vee \neg v_{20} \qquad \neg v_{11} \vee \neg v_{21} \qquad \neg v_{12} \vee \neg v_{22}$$

Une seule clause est nécessaire pour encoder la contrainte C_2 :

$$\begin{array}{lll} \neg v_{21} \vee \neg v_{30} \vee \neg v_{40} & \neg v_{21} \vee \neg v_{30} \vee \neg v_{42} & \neg v_{21} \vee \neg v_{31} \vee \neg v_{41} \\ \neg v_{21} \vee \neg v_{31} \vee \neg v_{42} & \neg v_{21} \vee \neg v_{32} \vee \neg v_{40} & \neg v_{21} \vee \neg v_{32} \vee \neg v_{41} \\ \neg v_{21} \vee \neg v_{32} \vee \neg v_{41} & & \end{array}$$

L'ensemble des clauses permettant de coder la contrainte C_3 sont :

$$\neg v_{40} \qquad \neg v_{41} \vee v_{30} \qquad \neg v_{42} \vee v_{30} \vee v_{31}$$
$$\neg v_{32} \qquad \neg v_{31} \vee v_{42} \qquad \neg v_{30} \vee v_{41} \vee v_{42}$$

La conjonction de ces clauses et celles permettant de modéliser les domaines des variables représente bien une formule `CNF` *équivalente pour la satisfiabilité au réseau de contrainte* N.

Dans ce chapitre nous nous sommes intéressés à la modélisation de problèmes à l'aide d'une conjonction de contraintes. Dans le chapitre suivant, nous présentons les algorithmes permettant de rechercher une ou plusieurs solutions d'un problème exprimer à l'aide de ces formalismes (`SAT` ou `CSP`).

Chapitre 2

De la recherche d'une solution au comptage de modèles

Sommaire

2.1 Résolution du problème SAT 24
2.1.1 Solveur SAT moderne : *CDCL* 24
2.1.2 La propagation unitaire 25
2.1.3 Analyses de conflits basées sur l'analyse du graphe d'implications 27
2.1.4 Heuristiques de branchement 28
2.1.5 Redémarrages 29
2.2 Résolution du problème CSP 29
2.2.1 Méthode de branchement binaire 30
2.2.2 Méthode de branchement non-binaire (ou n-aire) 31
2.2.3 Le filtrage 34
2.3 Compteurs de modèles 36
2.3.1 Le comptage exact de modèles 36
2.3.2 Le comptage approché de modèles 38
2.3.3 Le comptage de modèles via compilation de connaissances 39

Il est bien connu que déterminer si un réseau de contraintes possède une solution ou non est un problème NP-complet. Autrement dit, il n'existe pas d'algorithme déterministe en temps polynomial (sauf si P = NP) permettant de résoudre ce problème, mais nous sommes capables de vérifier en temps polynomial si une instanciation des variables est une solution ou non du problème. Les problèmes représentés par des réseaux de contraintes dont le but est de déterminer s'il existe ou non une solution sont appelés problèmes de satisfaction de contraintes. Résoudre un tel problème consiste souvent à trouver une (ou plusieurs) solution(s) au réseau de contraintes ou à prouver son incohérence.

Une façon de déterminer si un réseau de contraintes possède une solution consiste à se déplacer dans l'ensemble des affectations possibles des variables. Un algorithme naïf consisterait donc à essayer une à une l'ensemble des affectations possibles du réseau de contraintes et de vérifier si l'instanciation considérée est un modèle ou non. Il est clair qu'une telle approche n'est pas envisageable en pratique.

Pour résoudre ce problème, de nombreux algorithmes complets ont été proposés afin de vérifier la cohérence d'un problème exprimé à l'aide de contraintes. Ces algorithmes se divisent principalement en deux catégories regroupant chacune plusieurs familles d'algorithmes. La première catégorie regroupe les familles d'algorithmes syntaxiques. Ces derniers s'intéressent à la structure du problème en ne prenant pas en compte la valeur des variables (résolution [Rob65, Gal77], réécriture [Boo54, Sha40], etc.). La

seconde catégorie regroupe les familles d'algorithmes sémantiques qui, contrairement aux approches syntaxiques, s'attachent au sens des variables pour tenter de trouver une solution (énumération [Qui50, DLL62a, GB65, JW90], diagrammes de décision binaires [Ake78, Bry92, US94], etc.).

En pratique, les approches les plus utilisées sont les algorithmes énumératifs reposant le plus souvent sur une recherche en profondeur avec retour arrière non chronologique. Comme nous le verrons dans la dernière section, les compteurs de modèles sont aussi basés sur ces algorithmes [SBB+04a]. Cette méthode consiste à étendre de manière incrémentale une instanciation partielle en une instanciation complète représentant une solution au problème. Pour ce faire, nous affectons une à une les variables du réseau de contraintes jusqu'à trouver une solution. Lorsque l'instanciation partielle générée falsifie l'une des contraintes, un conflit est rencontré. et un retour arrière est réalisé afin de résoudre ce conflit et de réduire le nombre total d'assignations à vérifier. La trace de la recherche est généralement représentée sous la forme d'un arbre, appelé l'arbre de recherche. L'arbre complet représentant l'ensemble des instanciations possible est nommé l'espace de recherche (voir par exemple la figure 2.2).

2.1 Résolution du problème SAT

Comme précisé dans le chapitre précédent, SAT est le problème de décision consistant à déterminer si une formule sous forme normale conjonctive (CNF) possède ou pas un modèle. La cohérence d'une formule propositionnelle est le problème de décision de référence puisque ce fut le premier prouvé NP-complet. Il représente donc le problème NP-complet de référence [Coo71]. De plus, il a été montré que l'ensemble des problèmes NP-complets se traduisent naturellement en problèmes SAT ou peuvent être réduits à ce problème.

Dans cette section, nous présentons une approche complète permettant de résoudre le problème SAT. Pour une vue complète des différentes approches possibles pour résoudre SAT, le lecteur intéressé pourra se référer à [BBH+09]. Les approches complètes permettent en un temps fini de déterminer la cohérence de n'importe quelle formule CNF donnée. Il s'agit généralement de parcourir en profondeur un arbre de recherche, où chaque nœud correspond à l'assignation d'une variable à *vrai* ou à *faux*, et chaque chemin entre la racine et une feuille correspond à une interprétation de la formule. Le formalisme sous la logique propositionnelle pouvant être vu comme un cas particulier de celui des réseaux de contraintes, les algorithmes permettant de trouver une solution à un réseau de contraintes peuvent être appliqués à la logique propositionnelle. Puisque les algorithmes complets pour le problème SAT ne se basent que sur des formules sous forme normale conjonctive (CNF), les approches visant la logique propositionnelle sont modifiées afin de prendre en compte les particularités des formules CNF. Dans la suite, nous présentons les solveurs *CDCL* (*Conflict Driving Clause Learning*) [MMZ+01a] et les fonctions qui font que cet approche est efficace en pratique.

2.1.1 Solveur SAT moderne : *CDCL*

La procédure *CDCL* a été introduit par [MSS96] (et améliorée par [MMZ+01b]) et est une extension de la procédure *DPLL* [DLL62b]. Cette méthode tente de tirer parti de l'analyse de conflits afin d'apprendre une nouvelle clause et effectuer un retour arrière non chronologique. L'algorithme *CDCL* a deux avantages majeurs par rapport à l'algorithme *DPLL* : (i) effectuer un *backjumping* permet de ne pas considérer une partie de l'arbre de recherche ne possédant pas de solution et (ii) l'apprentissage de clauses permet d'éviter de considérer des sous-arbres de recherche, lesquels ont été montrés inconsistants lors d'une précédente analyse de conflits. L'algorithme *CDCL* ne pouvant pas être présenté facilement

de manière récursive, une version itérative de celui-ci est donnée dans l'algorithme 1.

Algorithme 1 : Solveur *CDCL*

Données : Σ une formule sous CNF
Résultat : SAT ou UNSAT

```
1  Δ ← ∅;                                          /* ensemble de clauses apprises */
2  I_p ← ∅;                                        /* interprétation partielle */
3  dl ← 0;                                         /* niveau de décision */
4  tant que (true) faire
5      α ← propagationUnitaire(Σ ∪ Δ, I_p);
6      si (α = null) alors
7          si (toutes les variables sont affectées) alors retourner SAT;
8          x ← heuristiqueDeBranchement(Σ ∪ Δ, I_p);
9          ℓ ←affectePolarité(x);
10         dl ← dl + 1;
11         I_p ← I_p ∪ {(l^dl)};
12         si faireReduction() alors reductionClausesApprises(Δ);
13     sinon
14         β ←analyseConflit(Σ ∪ Δ, I_p, α);
15         bl ←calculRetourArrière(I_p, bl);
16         si β = ⊥ alors retourner UNSAT;
17         Δ ← Δ ∪ β;
18         retourArrière(Σ ∪ Δ, I_p, bl);           /* mise à jour de I_p */
19         si (faireRedémarrage()) alors redémarrage(I_p, dl);
```

Typiquement, l'algorithme *CDCL* peut être assimilé à une séquence de décisions suivies de propagations des littéraux unitaires, jusqu'à l'obtention d'un conflit. Chaque littéral choisi comme littéral de décision (ligne 8) est affecté suivant une certaine polarité (ligne 9) à un niveau de décision donné (ligne 11). Si tous les littéraux sont affectés, alors $\mathcal{I}_p$ est un modèle de Σ (ligne 7). À chaque fois qu'un conflit est atteint par propagation (lignes 13–19), une clause β est calculée en utilisant une méthode d'analyse de conflits donnée et un niveau de *backjump* est calculé en fonction de la clause β (lignes 14–15). À ce stade, il est possible de prouver l'inconsistance (β est la clause vide) de la formule (ligne 16). Si ce n'est pas le cas, un retour arrière est effectué et le niveau de décision devient égal au niveau de *backjump* (ligne 18). Ensuite, certains solveurs *CDCL* forcent le redémarrage et dans ce cas, un retour arrière est effectué au sommet de l'arbre de recherche (ligne 19). Finalement, la base de clauses apprises peut être réduite lorsque celle-ci est considérée comme trop volumineuse (ligne 12).

Dans la suite, nous donnons une description des différentes fonctions laissées en suspens lors de la description.

2.1.2 La propagation unitaire

Commençons par présenter les différentes satisfactions des clauses par des interprétations. Soit α une clause, elle est :

— **satisfaite** par une interprétation $\mathcal{I}$ si au moins l'un de ses littéraux est affecté à vrai par $\mathcal{I}$;
— **falsifiée** par une interprétation $\mathcal{I}$ si tous ses littéraux sont affectés à faux par $\mathcal{I}$;
— **unitaire, binaire** ou **n-aire** si elles contient respectivement un, deux ou n littéraux différents ;

La propagation unitaire est la forme de filtrage la plus simple. Elle repose sur la propriété suivante : si l est un littéral d'une clause unitaire de Σ, alors les seules interprétations modèles de Σ doivent satisfaire

la clause unitaire l. En effet, une interprétation qui satisfait la négation du littéral conduit nécessairement à falsifier la clause unitaire et donc à une contradiction. Ainsi, la propagation unitaire consiste à supprimer toutes les clauses dans lesquelles le littéral l apparaît et à supprimer toute occurrence du littéral complémentaire, c'est-à-dire réduire les clauses contenant ce littéral complémentaire. Cette simplification est répétée jusqu'à ce que la formule ne contienne plus de clauses unitaires ou jusqu'à l'obtention d'une contradiction (une clause vide). Il est important de remarquer que cette simplification ne modifie pas la cohérence de la formule initiale. En effet, la formule obtenue après application de la propagation unitaire jusqu'à l'obtention d'un point fixe conjointe avec les littéraux produits est logiquement équivalente à la formule initiale.

Exemple 24 (Propagation unitaire) *Soit la formule* CNF Σ *suivante :*

$$x \wedge (x \vee y \vee z) \wedge (x \vee \neg u) \wedge (\neg z \vee \neg x) \wedge (\neg y \vee z \vee u)$$

Le littéral x étant le littéral d'une clause unitaire, la variable x est donc affectée à la valeur vrai. L'ensemble des clauses contenant x sont supprimées et la clause $\neg z \vee \neg x$ est simplifiée par la clause $\neg z$. Après propagation, nous obtenons la formule CNF *suivante : $\neg z \wedge (\neg y \vee z \vee u)$ $\neg z$ étant un littéral d'une nouvelle clause unitaire, la variable z est à sont tour affectée à faux. Le littéral z est donc supprimé de la clause $\neg y \vee z \vee u$. L'ensemble des clauses unitaire ont donc été propagées. La formule* CNF *$\neg y \vee u$ représente donc Σ après propagation. Remarquons que Σ est bien équivalente à la formule $x \wedge \neg z \wedge (\neg y \vee u)$.*

La propagation unitaire peut également être utilisée comme méthode de prétraitement pour extraire d'importantes informations de l'instance initial comme les littéraux impliqués, équivalents ou unitaires [Ber01, Nov03]. Elle permet aussi de déduire des portes logiques ou des sous-clauses dans la CNF initiale [GOMS04, PHS08].

Puisque la propagation unitaire est réalisée souvent elle consomme énormément de ressources de calcul. Pour de nombreux problèmes, elle induit à elle seule la grande majorité des affectations effectuées par le solveur. Puisque dans ce genre de situation 90% du temps consacré à la recherche d'une solutions est passé à effectuer ce traitement. Par conséquent, l'implémentation de cette méthode a été très étudiée. Dans la suite nous présentons une des avancées qui a permis au solveur SAT d'obtenir le succès que nous leur connaissons, à savoir les *watched literals*.

Les Watched Literals

Afin d'améliorer l'efficacité de la propagation unitaire de nombreuses méthodes ont été proposées, dont celle décrite dans [ZM02] qui représente incontestablement l'avancée la plus importante dans ce domaine. L'idée de cet algorithme est basée sur une structure de donnée dite « paresseuse ». Elle permet de traiter la propagation unitaire avec une complexité moyenne sous-linéaire. Le principe est de surveiller pour chaque clause de la formule deux littéraux x_i et x_j non affectés à *faux* par l'interprétation courante. Ces deux littéraux, appelés *watchs*, permettent de déterminer que la clause n'est ni unitaire, ni falsifiée. En effet, pour savoir si une clause est unitaire il suffit de vérifier si tous ses littéraux son affectés à *faux* sauf un. Une clause est falsifiée si tous ses littéraux sont affectés à *faux*. Lorsque les deux littéraux x_i et x_j ont été sélectionnés pour surveiller la clause, il n'est pas nécessaire d'observer cette clause à chaque affectation tant que ces littéraux n'ont pas été affecté. Ainsi, à chaque littéral est associé une liste de clauses à surveiller. Lorsque qu'un littéral x_i est affecté à *vrai*, il suffit alors de ne vérifier que la liste des clauses surveillées par le littéral complémentaire $\neg x_i$ et chercher un « remplaçant » pour chaque de ces clauses.

Lorsqu'un littéral x_i est affecté à *vrai* les clauses dans lesquelles il apparaît sont par définition satisfaites. Il n'est donc plus nécessaire de les surveiller. Nous devons donc simplement étudier les différents

cas pouvant se produire lorsqu'un littéral est affecté à *faux*. Considérons une clause $\alpha = x_1 \vee x_2 \vee \cdots \vee x_k$ dans laquelle les littéraux $x1$ et x_k sont les *watchs* de α. Nous distinguons trois cas susceptibles d'intervenir lors de la propagation unitaire après l'affectation du littéral $\neg x_1$:

- S'il existe un littéral x_i non affecté par l'interprétation courante et tel que $x_i \neq x_1$ et $x_i \neq x_k$, alors x_i peut surveiller la clause α (par exemple x_2) ;
- Si lors de la recherche d'un nouveau *watch* x_i, il existe un x_i qui est satisfait par l'interprétation courante (par exemple x_2), alors il n'est pas utile de remplacer le *watch* x_1. En effet, cela signifie que x_1 est affecté après x_i. Par conséquent lorsque x_i sera de nouveau non affecté, x_1 le sera aussi et il sera ainsi de nouveau à même de surveiller α ;
- Lorsqu'aucun littéral différent de x_k n'est affecté à *faux*, deux situations se présentent :
  - x_k n'est pas affecté, alors la clause est unitaire et le littéral x_k est alors propagé à *vrai* ;
  - x_k est affecté à *faux*, alors la clause est falsifiée par l'interprétation courante. Dans ce cas, la propagation unitaire conduit à un conflit.

  Dans les deux cas, il n'est pas nécessaire de modifier les *watchs* puisque lors d'un retour-arrière avant l'affectation de x_1, les deux littéraux, x_1 et x_k, seront en mesure de surveiller α.

Grâce à cette structure, nous remarquons que lors de l'affectation d'une variable, seul un sous-ensemble des clauses contenant une occurrence de cette variable est testé. Mais cela est suffisant pour réaliser les propagations unitaires durant l'affectation de valeurs lors de la recherche. Il est important de noter qu'aucune mise à jour n'est à effectuer lorsque le solveur effectue un retour-arrière puisque les *watchs* seront forcément des littéraux non affectés. L'utilisation de cette structure explique la rapidité du traitement de la propagation unitaire. Cependant l'utilisation de structures paresseuses possède quelques inconvénients. L'utilisation d'heuristiques « syntaxiques » n'est plus possible car celles-ci ont besoin d'une connaissance complète de l'instance. Ainsi, lorsque les solveurs *CDCL* ont été présentés, une heuristique ne s'appuyant pas sur la structure de la formule a été introduit. Cette heuristique, nommée *VSID*, est basée sur une des composantes les plus importantes des solveurs SAT modernes : l'analyse de conflit.

2.1.3 Analyses de conflits basées sur l'analyse du graphe d'implications

Les premiers travaux sur l'analyse de conflits ont été réalisés sur les CSP [Pro93]. Les interactions entre le monde des CSP et celui de SAT ont permis l'élaboration et le développement de ces techniques dans le cadre de SAT [MSS96, BJS97], qui, depuis, sont devenues incontournables.

Comme nous avons pu le souligner précédemment, le but de l'analyse de conflits est de trouver un ensemble de littéraux responsables d'une situation d'échec. Une fois cet ensemble de littéraux localisé une nouvelle clause est générée afin d'indiquer au solveur qu'il n'existe pas de solution dans un certain espace de recherche. À l'heure actuelle, le schéma d'analyse de conflits le plus couramment utilisé est basé sur l'analyse du graphe d'implications. Ce graphe est un graphe dirigé acylique (DAG) permettant de représenter les dépendances entre les clauses et les assignations obtenues par propagation des littéraux unitaires. Pour effectuer cela, un nœud est associé à chaque littéral affecté à vrai par l'interprétation courante. Ensuite, une arête entre un nœud x et un nœud y est créée si l'affectation de x à vrai a impliqué l'affectation de y à vrai.

Lorsque l'interprétation partielle est conflictuelle, le graphe d'implications généré comporte deux nœud représentant deux littéraux complémentaires. Dans ce cas, il est possible d'ajouter un nœud supplémentaire ($\perp$) symbolisant une situation conflictuelle. Lorsqu'un tel nœud est présent il est possible d'analyser le graphe afin d'extraire les littéraux responsables de ce conflit. Généralement, cette analyse est basée sur la notion d'*UIP* (*Unique Implication Point*). Un *UIP* est un nœud du dernier niveau de décision qui domine le conflit.

Remarque 5 *Les UIP peuvent être ordonnés en fonction de leur distance avec le conflit. Le premier*

point d'implication unique (first-UIP pour « First Unique Implication Point ») est l'UIP le plus proche du conflit tandis que le dernier UIP (last-UIP pour « Last Unique Implication Point ») est le plus éloigné, c'est-à-dire le littéral de décision affecté au niveau du conflit.

La localisation d'un *UIP* permet de fournir une décision alternative provoquant le même conflit. À l'heure actuelle, les démonstrateurs `SAT` modernes utilisent pour la plupart la notion de *first-UIP* afin d'extraire un *nogood* [MMZ+01b]. Cette clause est générée en effectuant des résolvantes entre les clauses responsables du conflit (utilisé durant la propagation unitaire) en remontant de celui-ci vers la variable de décision du dernier niveau jusqu'à obtenir une clause contenant un seul littéral du dernier niveau de décision (le *UIP*). Ce processus nommé preuve par résolution basée sur les conflits permet d'extraire une clause ne possédant qu'un seul littéral du dernier niveau. Cette clause est nommée clause *assertive*.

La clause assertive correspondant au *first-UIP* ainsi générée est apprise par le solveur et permet non seulement d'éviter à l'avenir d'atteindre à nouveau cet échec, mais aussi d'effectuer un retour arrière. En effet, la connaissance de cette clause falsifiée permet d'affirmer que le littéral assertif doit être propagé plus tôt dans l'arbre de recherche.

Pour terminer, à chaque conflit les solveurs *CDCL* apprennent une nouvelle clause dont la taille peut être relativement importante. Dès lors, ils se heurtent à deux problèmes majeurs : l'utilisation de la mémoire et le temps utilisé pour la propagation unitaire. Pour remédier à cela, la base de clauses apprises est régulièrement réduite (`reductionClausesApprises`). Le choix des clauses à conserver est déterminé en fonction des clauses utilisées lors de la génération des clauses conflits. C'est aussi de cette manière que sont pondérées les variables. Ce poids est à la base de l'heuristique de branchement *VSID*.

2.1.4 Heuristiques de branchement

Le choix de la prochaine variable à affecter possède une grande incidence sur la taille finale de l'arbre de recherche et par conséquent sur le temps d'exécution de l'algorithme. La taille de l'arbre de recherche peut varier de manière exponentielle selon l'ordre suivant lequel les variables sont affectées [LA97]. Malheureusement, trouver l'ordre des variables à affecter permettant de conduire à l'arbre de recherche minimal est un problème NP-difficile [Lib00]. Vu la complexité pour obtenir l'ordre de variables de branchement optimale, il semble raisonnable d'estimer quelle est la meilleur variable de branchement via une heuristique plutôt que de la calculer précisément. Pour être efficace, une telle heuristique doit permettre de réduire l'arbre de recherche tout en ayant un temps de calcul rapide. Une heuristique trop gourmande en temps, même si elle permet de fortement réduire la taille de l'arbre de recherche, peut se révéler moins performante qu'une heuristique plus rapide qui générera plus de branchements. Dans ce manuscrit, nous nous intéressons uniquement à l'heuristique de branchement la plus utilisée dans les implémentations modernes des solveurs `SAT` : l'heuristique *VSID*. Pour de plus ample informations, le lecteur pourra se référer à [BBH+09, Lag11].

L'heuristique *VSID* (« *Variable State Independent Decading Sum* »), proposée par [ZMMM01], est une approche de type rétrospective (*i.e.* de type « *look-back* ») qui cherche à éviter de futures situations d'échec. Le principe de ce type d'approche est de tirer parti des informations obtenues durant la recherche.

La méthode *VSID* associe donc un compteur à chaque variable du problème, appelé *activité*. Lorsqu'un conflit survient, une analyse de ce conflit est effectuée. Elle consiste à déterminer la raison du conflit et augmente l'activité des variables en étant la cause de cet échec. Au prochain choix, la variable ayant la plus grande activité est choisie comme prochaine variable de décision. Plus l'activité d'une variable est grande plus elle est impliquée dans les conflits, elle est donc considérée comme fortement contrainte. Le principe est de chercher à éliminer les variables se trouvant dans de nombreux conflits le

plus tôt possible dans la recherche et ainsi d'obtenir des conflits en haut de l'arbre de recherche. Afin de ne pas donner trop d'importance aux variables entrées en conflit au début de la recherche, l'activité des variables est régulièrement divisée par une constante. Cela permet de prendre en compte les variables apparaissant plus récemment dans les conflits.

2.1.5 Redémarrages

Comme observé par [GSCK00] les premiers choix effectués lors d'une recherche complète sont prépondérants. En effet, les auteurs ont montré expérimentalement qu'exécuter la même approche sur le même problème mais avec des choix initiaux différents conduit à des temps de résolution totalement hétérogènes. Ces expérimentations ont permis d'identifier un phénomène singulier nommé phénomène de longue traînée (« heavy tail ») [GW94, Wal99, GSCK00, CGS01]. Afin d'y remédier les auteurs proposent de redémarrer la recherche au bout d'un certain temps (« *restart* »). L'idée est que si la recherche échoue depuis un certain nombre de temps (évalué en nombre de retour arrière) alors il est jugé peu probable que la recherche aboutisse en un temps raisonnable. Dans ce cas, l'algorithme est relancé avec la formule initiale tout en conservant certaines informations (les scores pour *VSID* par exemple) ou les clauses apprises afin d'effectuer des choix plus judicieux au début de l'arbre.

Ces stratégies se décomposent en deux catégories. La première classe regroupe les stratégies de redémarrages statiques lesquelles sont établies au commencement de la recherche (intervalle fixe [Rya04, MMZ+01b, NGPH06], suite géométrique [Wal99] et intervalle variable [LSZ93, Bie08b, PD07, LH05], ...). La seconde classe regroupe les approches dynamiques qui tentent d'exploiter différentes informations en cours de recherche afin de déterminer si le solveur doit effectuer ou pas un redémarrage (variation de phase [Bie08a], `glucose` [AS09], hauteur des sauts [HJS10], ...).

2.2 Résolution du problème CSP

La manière dont la résolution est réalisée dans le cadre du problème CSP est assez similaire au cadre SAT. À chaque itération, une variable est sélectionnée par l'algorithme et un branchement est effectuée sur une valeur de son domaine qui est alors réduit. Cette étape est appelée étape de décision. Dans cette thèse nous nous sommes restreints à la méthode de branchement suivante, cependant de nombreuses autres approches de branchement pour le problème CSP ont été étudiées.

Définition 29 (Étape de décision) *Soit $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes et V_i une variable de $\mathcal{V}$. Une décision correspond soit à l'assignation d'une variable V_i à la valeur v, soit à la réfutation de la variable V_i à la valeur v.*

Si le branchement est une assignation, alors le domaine de la variable est réduit à un singleton. Sinon, c'est une réfutation et la valeur choisie est supprimée du domaine. Enfin, un test pour vérifier qu'aucune contrainte n'est violée est effectué. Lorsqu'une contrainte est devenue incohérente, la dernière modification sur le réseau est mise en cause, elle est supprimée par un retour-arrière sur la variable qui vient d'être affectée.

Après avoir réduit les domaines de toutes les variables du réseau de contraintes à des singletons, si toutes les contraintes sont satisfaites alors une solution est trouvée. Cette solution correspond à l'instanciation courante. Elle est représentée par l'ensemble des affectations effectuées sur le chemin de l'arbre entre la racine et la feuille solution. Si l'arbre de recherche est entièrement exploré et qu'aucune solution n'a été trouvée, alors l'algorithme a démontré que le réseau de contraintes donné est incohérent. La construction de l'arbre de recherche peut se faire suivant deux méthodes de branchement différentes. La méthode de branchement binaire ou le branchement non binaire. Chaque nœud de l'arbre de recherche correspond à une variable choisie par l'algorithme et chaque branche correspond à une étape de décision.

2.2.1 Méthode de branchement binaire

Dans le cas de branchement binaire, l'étape de décision correspond à des assignations et des réfutations. Lors d'une itération, l'étape de sélection d'une variable est remplacée par la sélection d'une paire variable-valeur (V_i, v_j) où V_i est une variable encore non affectée et v_j une valeur de son domaine courant. Deux cas sont ainsi considérés pour le branchement, d'un côté la décision sera l'assignation $V_i = v_j$ et de l'autre la réfutation $V_i \neq v_j$. Remarquons que dans le cas où l'assignation est considérée, la variable V_i est assignée est ne sera donc plus sélectionnée par l'algorithme pour apparaître dans une décision de la branche courante.

À chaque décision, le réseau de contraintes initial $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ est modifié par les assignations ou réfutations de valeurs effectuées. Lorsque la décision est une assignation du type $V_i = v_j$, alors le domaine D_i de la variable V_i est restreint au singleton $D_i = \{v_j\}$. Lors d'une réfutation du type $V_i \neq v_j$, la valeur v_j est supprimée du domaine D_i de la variable V_i, noté $D_i = D_i \setminus \{v_j\}$. À la suite de chaque décision, un sous-réseau de contraintes peut être calculé. Le sous-réseau de contraintes obtenu sera au moins aussi succinct que le réseau de contraintes initial. Nous verrons dans la suite de ce chapitre que la modification du domaine des variables peut entraîner la simplification, voir la suppression, de certaines contraintes du réseau initial. Le sous-réseau obtenu après la décision est noté $N|_{V_i=v_j}$ ou $N|_{V_i\neq v_j}$ selon le cas.

L'algorithme 2 génère un arbre de recherche binaire. La fonction contrainteFalsifiee, ligne 1, retourne vrai si au moins l'une des contraintes du réseau passé en paramètre n'est pas satisfiable et faux sinon. Si au moins l'une des contraintes du réseau est falsifiée, alors ce sous-réseau est incohérent et $faux$. Sinon, ligne 2, si toutes les variables sont assignées, alors le réseau de contraintes possède une solution et $vrai$ est retourné. Sinon, un nouveau couple variable-valeur est sélectionné ligne 3. Enfin, ligne 4 la valeur est dans un premier temps assignée à la variable et l'algorithme est appelé récursivement avec le sous-réseau de contraintes obtenu, puis il est fait de même avec la réfutation de cette valeur pour la variable V_i.

Algorithme 2 : solveurBinaire(N)

entrée : un réseau de contraintes N
sortie : $vrai$ s'il existe une solution, $faux$ sinon

1 si contrainteFalsifiee(N) alors return $faux$;
2 si $\forall V_i \in \mathcal{V}, |D_i| = 1$ alors return $vrai$;
3 choisir un couple (V_i, v_j) telle que $|D_i| > 1$ et $v_j \in D_i$;
4 return solveurBinaire($N|_{V_i=vj}$) $\vee$ solveurBinaire($N|_{V_i\neq vj}$) ;

Considérons un problème de coloriage comportant 10 variables ayant comme domaine les valeurs $\{Rouge, Bleu, Vert\}$. La figure 2.1 illustre un arbre de recherche binaire partiel correspondant à ce problème. Cet arbre est le résultat partiel d'un algorithme à branchement binaire comme l'algorithme 2. Chaque nœud représente le couple sélectionné par l'algorithme. Les branches de gauche de l'arbre de recherche représentent les cas où la valeur est assignée à la variable et la branche de droite les cas où la valeur est réfutée. Dans cet exemple, le couple $(V_1, Rouge)$ a été sélectionné et la valeur $Rouge$ a été assignée à la variable V_1. Il en est de même avec le couple $(V_2, Bleu)$, l'algorithme assigne la valeur $Bleu$ à V_2 et continue de construire récursivement le sous-arbre correspondant en choisissant un nouveau couple (V_i, v_j). Une fois le sous-arbre entièrement exploré, si il n'y a pas de solution la valeur $Bleu$ de V_2 et le sous-arbre correspondant à la séquence d'affectation $< (V_1 = R), (V_2 \neq B) >$ est exploré. Ainsi de suite pour l'ensemble des couples possibles jusqu'à trouver une solution si elle existe ou démontrer l'incohérence du réseau en explorant entièrement l'arbre de recherche.

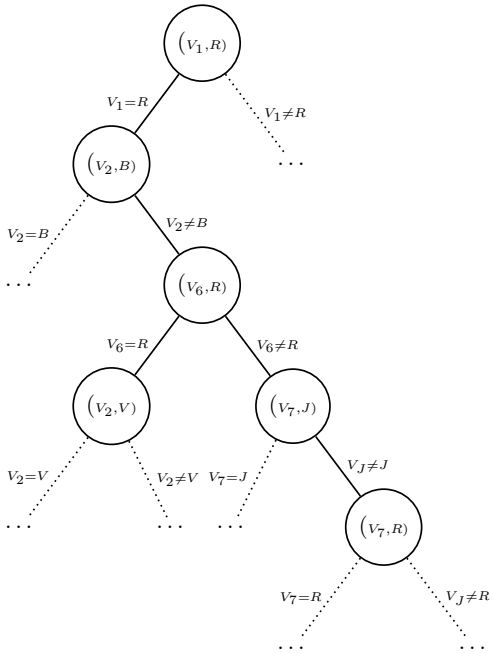


FIGURE 2.1 – Arbre binaire de recherche partiel du problème de coloriage obtenu par un algorithme à branchement binaire.

2.2.2 Méthode de branchement non-binaire (ou n-aire)

Dans le cas d'un branchement n-aire, l'étape de décision correspond à une assignation. Lors d'une itération de l'algorithme de recherche, une variable qui est encore non assignée est sélectionnée. Nous essayons alors d'assigner successivement l'ensemble des valeurs du domaine de la variable choisie. Ceci correspond à un nœud de l'arbre de recherche. Chaque branche du nœud représente l'assignation de la variable étiquetant ce nœud par une valeur de son domaine. Lorsqu'une assignation falsifie au moins une contrainte du réseau, cette branche n'est pas explorée et une autre valeur est assignée à la variable. Si toutes les valeurs de la variable falsifient au moins une contrainte, alors un retour arrière est effectué et le dernier choix de valeur pour la variable précédente est remis en cause. L'algorithme procède ainsi récursivement jusqu'à ce que soit toutes les variables soient assignées à une valeur, et dans ce cas une solution est trouvée, soit l'ensemble de l'arbre a été exploré et conduit toujours à un échec. Dans ce dernier cas, le réseau de contrainte est démontré incohérent.

Algorithme 3 : solveur-n-aire(N)

entrée : un réseau de contraintes N
sortie : $vrai$ s'il existe une solution, $faux$ sinon

```
1 si contrainteFalsifiee(N) alors return faux;
2 si ∀V_i ∈ 𝒱, |D_i| = 1 alors return vrai;
3 choisir une variable V_i telle que |D_i| > 1;
4 pour tous les v_j ∈ D_i faire
5     si solveur-n-aire(N|_{V_i=v_j}) alors return vrai;
6 return faux ;
```

L'algorithme 3 décrit un solveur utilisant un branchement n-aire. La fonction contrainteFalsifiee, ligne 3, retourne vrai si au moins l'une des contraintes du réseau passé en paramètre n'est pas satisfiable et faux sinon. Les deux premières lignes représentent les deux cas d'arrêt de la récursion. Soit l'une des contraintes du réseau est falsifiée, alors cette branche conduit à des sous-réseaux incohérents, $faux$ est donc retourné. Soit toutes les variables du réseau de contraintes sont assignées, ligne 1, alors le réseau possède au moins une solution et $vrai$ est retourné. Ensuite commence l'étape de décision. Tout d'abord, ligne 2 une variable encore non assignée est choisie, ce qui correspond à un nouveau nœud de l'arbre de recherche. Puis, ligne 3, nous assignons successivement toutes les valeurs du domaine de la variable et le réseau de contraintes est modifié pour prendre en compte cette décision. Ceci représente une nouvelle branche de l'arbre de recherche. L'algorithme est appelé récursivement avec le sous-réseau de contraintes correspondant, ligne 4. Si une solution est trouvée la recherche est stoppée et l'information est remontée aux parents du nœud. En cas d'échec, l'assignation est remise en cause et une autre valeur appartenant au domaine de la variable est assignée et ainsi de suite pour toutes les valeurs de son domaine. Enfin, si après avoir assigné l'ensemble des valeurs possibles aucune solution n'a été trouvé, alors ce sous réseau est démontré incohérent. La valeur $faux$ est alors retournée afin de remettre en cause l'assignation précédente.

La figure 2.2 représente un arbre de recherche partiel obtenu par une méthode de branchement n-aire.

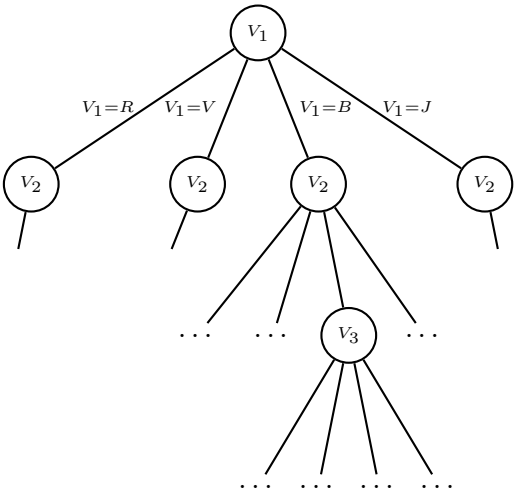


FIGURE 2.2 – Arbre de recherche avec branchement n-aire.

Hwang et Mitchell, dans [HM05], ont montré que ces deux méthodes de branchements ne sont pas équivalentes. La méthode de branchement binaire est plus efficace que la méthode non binaire principalement pour démontrer l'incohérence d'un réseau de contraintes. Cette thèse se limitant à la compilation de bases de connaissances décrites en réseaux de contraintes, nous nous sommes focalisés sur les schémas de branchements n-aires. En effet, la compilation ne possède un intérêt que si les bases de connaissances sont cohérentes.

Comme pour SAT, un aspect très important pour la recherche efficace de solution est le choix de la variable à assigner. En effet, le choix de la variable sélectionnée est cruciale pour la taille de l'arbre de recherche final obtenu. Une variable influant beaucoup sur le réseau permettra de considérablement réduire l'espace de recherche si elle est sélectionnée tôt dans l'arbre plutôt que vers la fin de la recherche. À la fois l'ordre des variables à sélectionner et l'ordre des valeurs à leur assigner sont importants pour une recherche efficace de solution. Ce choix peut améliorer sensiblement l'efficacité de la recherche et joue un rôle important en ce qui concerne la taille de l'arbre de recherche obtenu [BvR95, GMP+96, GFHT90,

HE80]. Cependant, trouver la meilleure variable à sélectionner pour un réseau de contrainte donné est aussi complexe que de trouver si ce réseau est cohérent [Lib00]. Afin d'aider les algorithmes à faire de « bons » choix, différentes heuristiques ont été proposées afin de guider la recherche. À chaque étape de décision, l'heuristique évalue la variable qui est considérée comme la meilleure ainsi que la meilleure valeur potentielle à assigner à cette variable à cette étape de la recherche. Un solveur CSP possède donc deux heuristiques de choix. Une première pour sélectionner la variable puis celle permettant de choisir la valeur à assigner à cette variable. Choisir la valeur la plus appropriée peut se révéler être une tâche complexe. Souvent le choix de la valeur est effectué de manière aléatoire ou en fonction d'un ordre prédéfini lié aux entiers codant les valeurs du domaine.

Choix de variable

Il est possible de fixer statiquement à l'avance l'ordre selon lequel les variables seront choisies, c'est-à-dire que cette ordre est défini avant le début de la recherche. Il est malheureusement difficile de prévoir l'aspect de l'arbre de recherche que nous obtiendrons, une telle approche se révèle donc peu efficace en pratique. Nous avons donc plus souvent recours à des heuristiques dynamiques. Ces heuristiques exploitent les informations sur l'état courant ou passé de la recherche afin de sélectionner la variable ayant le plus gros potentiel. La plupart de ces heuristiques s'appuient sur l'un des deux principes suivants. Soit sur le principe *first fail* [HE80] qui cherche à prendre des décisions de sorte d'arriver le plus vite possible vers un échec. Soit sur le principe *promise* [BPW03] qui tente d'assigner le maximum de variables afin de trouver un modèle. Ces deux principes d'heuristiques sont décrit et comparés dans [Wal06, BPW04]. Les heuristiques de choix de variables les plus connues sont :

- **L'heuristique dom** : cette heuristique ([HE80]) est la première heuristique dynamique ayant été proposée. La taille du domaine courant de chaque variable est utilisé pour déterminer la prochaine variable à sélectionner. Les variables sont ordonnées de façon croissante selon la taille courante de leur domaine. Puis, une variable ayant le plus petit domaine est choisie pour être la prochaine variable à assigner ;
- **L'heuristique ddeg** : cette heuristique ([DM89]) ordonne les variables de manière décroissante en fonction de leur degré courant. Le degré d'une variable correspond au nombre de contraintes la liant aux autres variables du problèmes à un instant donné de la recherche. Autrement dit, cette heuristique sélectionne en priorité une variable connectée avec le plus grand nombre de variables encore non assignées ;
- **L'heuristique wdeg** : cette heuristique ([BHLS04]) dirige la recherche vers les parties difficiles du problème en suivant le principe du *first fail*. Elle consiste à choisir les variables apparaissant le plus souvent dans les contraintes falsifiées. Elle va chercher à satisfaire en priorité les contraintes qui conduisent le plus souvent à des conflits. Un compteur de conflits est associé à chaque contrainte du problème. Celui-ci est initialisé à 1 au début de la recherche et est incrémenté lorsque la contrainte qui lui est associée est falsifiée pendant la recherche. Grâce à ces compteurs, il est possible de déterminer où sont les parties difficiles d'un réseau de contraintes et de diriger la recherche vers cette partie. Pour ce faire, le degré pondéré des variables du problème est calculé à partir de ces compteurs. Le degré pondéré d'une variable correspond à la somme des compteurs de conflits des contraintes impliquant cette variable et au moins une autre variable non assignée. Enfin, une variable ayant le plus grand degré pondéré est choisie ;
- **Combinaison d'heuristiques** : les trois heuristiques précédentes peuvent être combinées afin d'obtenir de nouvelles heuristiques. Ces heuristiques sont montrées plus performantes en pratique lorsqu'elles sont combinées que lorsqu'elles sont considérées séparément. L'heuristique *dom/ddeg* ([SG97]), par exemple, est en moyenne plus efficace que les heuristiques *dom* ou *ddeg* seules. Elle consiste à sélectionner en priorité une variable ayant le plus petit ratio taille du

domaine courant sur degré courant. La combinaison d'heuristiques *dom/wdeg* apparaît comme étant l'heuristique la plus performante par rapport aux autres combinaisons ([BHLS04, LBH04, HO05]). Elle consiste à choisir en priorité une variable ayant le plus petit ratio taille du domaine courant sur degré pondéré courant.

Dans le but de réduire l'espace de recherche à analyser, une idée est de diminuer le nombre d'affectations possibles à effectuer. Pour cela, les domaines des variables sont réduits pendant la recherche. Il est, en effet, possible de filtrer les domaines des variables, et donc d'en supprimer des valeurs, en tenant compte des propriétés des contraintes. Chaque nœud de l'arbre de recherche peut être associé à un sous-réseau de contraintes où un processus de filtrage a été réalisé sur l'ensemble du réseau de contraintes.

2.2.3 Le filtrage

Un propagateur assurant une forme de cohérence locale est un ingrédient incontournable d'un solveur pour les réseaux de contraintes. La propagation est en effet une étape effectuée avant même de débuter la recherche de solution et que nous continuons de faire pendant la recherche de solution. Cette étape est aussi appelée filtrage. Le but du filtrage est d'éliminer tout élément dont nous sommes assurés qu'ils ne peuvent figurer dans une solution du réseau de contrainte. Les propagateurs consistent à supprimer les valeurs des domaines pour qui, si ces valeurs sont assignées aux variables, une contradiction survient. On appelle cela une incohérence locale.

Le filtrage permet de supprimer les valeurs inutiles des domaines et donc de réduire la taille de l'espace de recherche à explorer lorsque le solveur cherche une solution. Le but du filtrage est de supprimer les valeurs qui ne peuvent pas faire partie d'une solution au problème. Dans cette thèse, nous nous restreignons aux notions de cohérence qui enlèvent uniquement des valeurs dans les domaines des variables. Les algorithmes de filtrage étant essentiels pour obtenir un bon solveur, ceux-ci ont été très étudiés par la communauté. Parmi les formes de cohérence locale, nous pouvons mentionner les trois plus populaires et les plus étudiés : la cohérence d'arcs (AC) et l'ensemble de ses dérivés, la cohérence de chemins, la k-cohérence. Pour de plus amples informations le lecteur peut se référer à [Bes06]. Nous définissons formellement la notion de cohérence locale de la manière suivante :

Définition 30 (Cohérence locale) *Soit $C = (S, R)$ une contrainte, et $V_i \in S$. Une valeur $d_j \in D_i$ est cohérence avec C si et seulement s'il existe un tuple $\mathcal{T}$ sur les variables de la portée S de la contrainte, tel que $d_j = \mathcal{T}(V_i)$ et que $\mathcal{T}$ satisfait la contrainte. Les valeurs des autres variables du tuple forment le* support *de d_j pour la contrainte C.*

Les méthodes de filtrage consistent à vérifier si l'instanciation courante est cohérente localement. Pour ce faire, les valeurs du domaine d'une variable qui ne peuvent pas satisfaire une contrainte sont supprimées.

Cohérence de nœud Ce filtrage est la forme de cohérence la plus simple puisqu'elle prend en compte les variables séparément. Elle ne concerne que les contraintes unaires. Elle consiste à vérifier que toutes les valeurs des variables de la portée des contraintes unaires sont cohérentes. Si l'une des valeurs falsifie une contrainte unaire, elle est supprimée du domaine de la variable.

Exemple 25 *Soient V_1 une variable de domaine $D_1 = \{1, 2, 3, 4\}$ et la contrainte unaire $C_1 = (V_1 > 1)$. Après filtrage par cohérence de nœud le domaine de V_1 est réduit à $D_1 = \{2, 3, 4\}$.*

Une fois cette forme de filtrage appliquée au réseau de contraintes, le réseau de contraintes est dit 1-cohérent. Lorsque les domaines ont été filtrés, les contraintes unaires peuvent être supprimées du réseau. En effet, cette méthode supprimant toutes les valeurs pouvant rendre incohérentes les contraintes unaires, par la suite ces contraintes seront toujours satisfaites.

Forward Checking Le *Forward Checking* (FC) est un algorithme de cohérence faible. Cette méthode de filtrage fut dans un premier temps proposée pour les réseaux de contraintes binaires par [HE80], puis étendue au cas des réseaux d'arité supérieure par [BMFL99]. Lorsqu'un domaine D_i d'une variable V_i du réseau de contrainte est un singleton v_{ij}, le *Forward Cheking* supprime toutes les valeurs du domaine des variables apparaissant dans une même contrainte $C = (S, R)$ que V_i, qui contredisent la relation R de C. Aucune opération n'est faite quand le domaine n'est pas un singleton.

Exemple 26 *Soit le réseau de contraintes $\mathcal{C}$ avec trois variables V_1, V_2 et V_3 ayant respectivement pour domaines $\{2\}$, $\{2, 3\}$ et $\{1, 2, 3, 4\}$, et les trois contraintes $C_1 = V_1 \neq V_2$, $C_2 = V_1 \neq V_3$ et $C_3 = V_2 \neq V_3$. Dans un premier temps, V_1 étant le seul domaine singleton, FC va sélectionner la contrainte C_1 et va supprimer la valeur 2 du domaine de V_2. FC va ensuite sélectionner la contrainte C_2 et supprimer la valeur 2 du domaine de V_3. Enfin, V_2 étant devenu un singleton, l'opérationest effectuée à nouveau avec C_1 et C_3 et 3 est supprimé du domaine de C_3. Après le Forward Checking nous avons donc respectivement pour chaque variables V_1, V_2 et V_3 les domaines $\{2\}$, $\{3\}$ et $\{1, 4\}$*

Cohérence aux Bornes La cohérence aux bornes (BC) consiste à ne chercher la cohérence locale uniquement sur les bornes du domaine des variables. Une contrainte $C = (S, R)$ est cohérente aux bornes si et seulement si $\forall V_i \in S, D_i \neq \varnothing$ et les valeurs $min(D_i)$ et $max(D_i)$ sont cohérentes avec C.

Exemple 27 *Étant donné le réseau de contraintes $\mathcal{C}$ de l'exemple précédent, $BC(\mathcal{C})$ va supprimer la valeur 2 du domaine de V_2 par la contrainte C_1. Enfin, les contraintes C_2 et C_3 sont cohérentes pour chaque valeur aux bornes du domaine des variables. Après le filtrage par cohérence aux bornes nous avons donc respectivement pour chaque variables V_1, V_2 et V_3 les domaines $\{2\}$, $\{3\}$ et $\{1, 2, 3, 4\}$*

Arc cohérence L'arc cohérence (AC) est une méthode de filtrage plus complète que celles citées précédemment mais aussi plus coûteuse. Elle est la méthode de cohérence locale la plus étudiée. Cette méthode de filtrage fut d'abord introduite dans le cas des réseaux de contraintes binaire par [Mac77] et étendue ensuite aux réseaux de contraintes n-aires. Nous définissons l'arc cohérence dans le cas général, c'est-à-dire pour un réseau de contraintes quelconque. Dans ce cas, l'arc cohérence est souvent dite arc cohérence généralisée (GAC). Le principe de ce filtrage est de garantir que chaque valeur du domaine d'une variable est cohérente avec les contraintes du réseau. Il n'est, en effet, pas toujours possible de trouver un support pour l'ensemble des valeurs du domaine d'une variable. Une contrainte $C = (S, R)$ est arc cohérent si et seulement si pour toute variable $V_i \in S$, $D_i \neq \varnothing$ et $\forall d_j \in D_i$, d_j est cohérente pour R. Si toutes les valeurs du domaine D_i de la variable V_i sont arc cohérentes (généralisées) avec C, alors V_i est arc cohérente (généralisée) pour C. Un réseau de contraintes $N = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ est arc cohérent généralisé si et seulement si $\forall V_i \in \mathcal{V}$ et $\forall C_i \in \mathcal{C}$, V_i est arc cohérent généralisé pour C_i.

Exemple 28 *Étant donné le réseau de contraintes $\mathcal{C}$ de l'exemple précédant, $AC(\mathcal{C})$ va supprimer la valeur 2 du domaine de V_2 par la contrainte C_1, la valeur 2 de V_3 par la contrainte C_2 et la valeur 2 de V_3 par la contrainte C_3. Après l'arc cohérence, nous avons donc respectivement pour chaque variables, V_1, V_2 et V_3 les domaines $\{2\}$, $\{3\}$ et $\{1, 4\}$.*

L'arc cohérence est souvent la méthode de filtrage implantée dans les solveurs CSP. Le fait de maintenir l'arc cohérence tout au long de la recherche a été proposé dans [SF94] et cette méthode est appelée MAC (*Maintening Arc Consistency*).

D'autres mécanismes de filtrages ont été proposés, certains beaucoup plus puissants que l'arc cohérence. Cependant, la plupart de ces techniques possède une complexité spatiale et temporelle trop élevée pour être calculées à chaque étape de la recherche. Nous pouvons par exemple citer l'arc cohérence

singleton (SAC) [DB97], la cohérence de chemins [Mon74] et la (i-j) cohérence [Fre85]. Le lecteur intéressé par ces méthodes de filtrages peut se référer à [Bes06]. Ces algorithmes de filtrages ont permis une forte amélioration des performances des solveurs. Ceux-ci s'appliquent de manière générale sur l'ensemble des contraintes du réseau.

2.3 Compteurs de modèles

Dans cette section, nous présentons le comptage de modèles et les différentes façons de l'aborder. Nous discutons des principaux algorithmes permettant de compter de le nombre exact de modèles d'une formule CNF. Ces algorithmes sont principalement basés sur la trace de solvers DPLL [SBB+04b, Thu06]. Pour plus d'informations sur le sujet, le lecteur pourra se référer à [GSS09].

Le problème du comptage de modèles, aussi appelé #SAT, consiste à déterminer le nombre de modèles d'une formule propositionnelle quelconque. Autrement dis, de calculer le nombre d'interprétations distinctes qui évaluent la formule à vrai. Ce problème est le problème #P-complet de référence. Il revient à compter le nombre de solutions possibles d'une formule.

Définition 31 (Classe #P) *La classe de complexité #P représente les problèmes qui comptent le nombre de solutions d'un problème de décision qui est dans la classe NP.*

Clairement, un problème dans la classe #P-complet est au moins aussi difficile qu'un problème dans la classe **NP**-complet. En effet, puisque si nous connaissons le nombre de solutions d'une instance donnée, nous sommes en mesure de savoir s'il en existe au moins une. Plus précisément, le théorème de Toda [Tod89] permet de définir qu'un algorithme en temps polynomial disposant d'un oracle de la classe #P peut résoudre n'importe quel problème de la hiérarchie polynomiale PH, autrement dit $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$. Plus précisément, cet algorithme n'a besoin que d'un appel à cet oracle #P pour résoudre un problème de la hiérarchie polynomiale.

Le comptage de modèles est un problème très intéressant car il apparaît dans beaucoup d'applications, telles la planification et la configuration. Cependant, le théorème de Toda donne une bonne indication quand à la difficulté des problèmes #P-complets.

2.3.1 Le comptage exact de modèles

Le problème du comptage de modèles peut être résolu en pratique en étendant l'un des algorithmes les plus populaire pour SAT, l'algorithme DPLL. Ces algorithmes étant étudiés pour le problème SAT, nous avons pour habitude de calculer les modèles de formules CNF. C'est un algorithme de recherche systématique qui recherche dans l'espace des interprétations jusqu'à trouver un modèle à la formule ou montrer qu'il n'existe pas de modèle pour cette formule. En particulier, étant donnée une formule CNF Σ, l'algorithme DPLL choisit une variable x_i de Σ et est appelé récursivement en considèrent la formule $\Sigma' = \Sigma \mid x_i$, puis $\Sigma' = \Sigma \mid \neg x_i$. Enfin, il décide si la formule est cohérente ou non lorsque $\Sigma' \models \top$ ou que $\Sigma' \models \bot$. Cette approche peut facilement être étendue pour calculer le nombre exact de modèles d'une formule CNF. Il suffit de ne pas arrêter la recherche au premier modèle trouvé. En continuant la recherche ainsi, on cherche l'ensemble des autres modèles de la formule. Il suffit d'augmenter la valeur d'un compteur à chaque feuille $\top$ rencontrée dans l'arbre de recherche pour obtenir le nombre de modèles de la formule CNF. Ceci nous donne un algorithme naïf pour compter le nombre exact de modèles d'une formule.

La figure 2.3 représente un arbre de recherche complet pour la formule CNF $\Sigma = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$ portant sur les trois variables $\{x_1, x_2, x_3\}$. Cet arbre de recherche montre l'ensemble des chemins parcourus lors de la recherche exhaustive par l'algorithme DPLL sur la

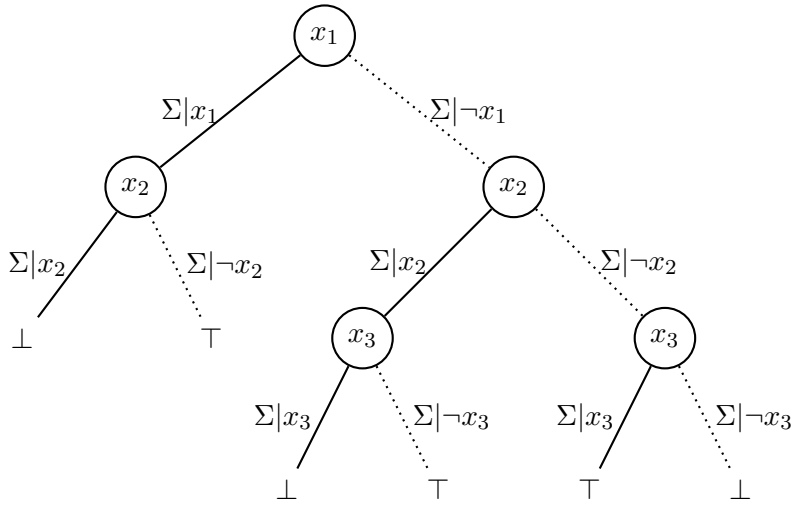


FIGURE 2.3 – Arbre de recherche DPLL

formule Σ. Le chemin entre la racine et un nœud représente une interprétation (partielle) des variables de Σ. Chaque feuille représente le résultat de la recherche lorsque l'interprétation représentée par le chemin entre la racine et la feuille est appliquée à la formule booléenne. Quand la feuille est étiquetée par $\bot$, l'interprétation partielle n'est pas un modèle de la formule, et est un modèle quand elle est étiquetée par $\top$. Le nombre de modèles d'une feuille $\top$ est de 2^{n-t} où n est le nombre de variables de Σ et t le nombre de variables assignées dans l'interprétation partielle calculée. Le nombre de modèles de la CNF Σ correspond donc à la somme des nombres obtenus à chaque feuille $\top$, c'est-à-dire 4 modèles. Nous appelons cet arbre la trace de la recherche exhaustive effectuée par l'algorithme DPLL.

La trace de la méthode DPLL peut être interprétée comme une formule propositionnelle NNF qui satisfait le déterminisme et la décomposabilité [Dar04]. En effet, la trace obtenue lors de la recherche correspond au langage des DT (cf. chapitre 3). De plus, lorsque nous ajoutons à la méthode DPLL la reconnaissance de composantes connexes pendant de la recherche, la trace correspond au langage EDT.

L'algorithme DPLL étant conçu pour la recherche d'une solution, et non de l'ensemble des solutions, il ne peut pas être utilisé tel quel. Par exemple l'heuristique de choix de variable d'un algorithme DPLL va chercher à obtenir le plus rapidement possible un modèle, tandis qu'un compteur de modèle doit parcourir tout l'arbre de recherche. Ceci réduit fortement l'efficacité du solver. Plusieurs méthodes ont été développées afin d'améliorer cet algorithme. Les principales techniques utilisées par les algorithmes pour compter le nombre de modèles de Σ résident dans la conservation de la trace du DPLL, la décomposition de la formule pendant la recherche DPLL et enfin, la technique de caching.

L'analyse de composantes connexes

Elle permet de découper la formule en plusieurs sous-formules, sur lesquelles l'algorithme travaille séparément. Cela permet de réduire fortement l'espace de recherche et par conséquent le temps de recherche, puisque la recherche est effectuée sur une sous-formule avec un plus petit nombre de variables. De plus, l'approche peut être lancée en parallèle sur chaque sous-formule, ce qui permet un gain de temps non négligeable. La recherche de composantes connexes se fait depuis le graphe de contraintes de la formule CNF. Soit $\mathcal{G}$ la graphe de contraintes d'une formule CNF $\mathcal{F}$. Les sommets de $\mathcal{G}$ sont les variables de $\mathcal{F}$ et une arrête entre deux sommets signifie que les deux variables correspondantes apparaissent dans une même clause de $\mathcal{F}$. Pour découper la formule $\mathcal{F}$ en sous-formules indépendantes, il

suffit de calculer les composantes connexes $\mathcal{G}_1, \ldots, \mathcal{G}_n$ du graphe $\mathcal{G}$. Autrement dit, $\mathcal{G}$ est découpé en plusieurs sous-graphes tel que pour chaque sommet d'une composante connexe $\mathcal{G}_i$ il n'existe pas d'arête entre ce sommet et une autre composante connexe $\mathcal{G}_j$ avec $i \neq j$. De cette manière nous découpons $\mathcal{F}$ en plusieurs sous-formules $\mathcal{F}_1, \ldots, \mathcal{F}_n$, de telle sorte qu'un couple de variables de deux composantes connexes différentes n'apparaissent pas dans une même clause. Chaque sous-formule $\mathcal{F}_i$ ne contient que les clauses contenant les variables apparaissant dans les sommets de la composante connexe correspondante $\mathcal{G}_i$. Comme les composantes connexes sont disjointes, une clause n'apparaît que dans une seule sous-formule. Ainsi, nous pouvons calculer séparément le nombre de modèles de chaque sous-formules $\mathcal{F}_i$. Le nombre de modèles de la formule initiale $\mathcal{F}$ peut lui être calculé facilement en multipliant le nombre de modèles obtenu pour chaque sous-formules. Autrement dit, le nombre de modèles de la formule est $\#\mathcal{F} = \#\mathcal{F}_1 \times \cdots \times \#\mathcal{F}_n$ Avec l'approche DPLL, après chaque affectation d'une variable, des simplifications peuvent être effectuées. En effet, chaque clause satisfaite par les variables affectées peuvent être supprimées de la formule, ce qui simplifie de le graphe de contrainte. La recherche de composantes connexes peut donc être effectuée à chaque nœud du parcours de l'espace de recherche.

Le caching

Il permet d'éviter de recalculer la trace d'une recherche lorsque nous tombons sur une formule équivalente à une formule déjà calculée pendant la recherche. En effet, pendant l'exploration de l'espace de recherche, il est possible qu'à la suite des affectations et simplifications de la formule, nous tombions sur une sous-formule que nous avons déjà explorée précédemment dans l'arbre de recherche partiel. Lorsqu'une telle formule est trouvée, plutôt que tout recalculer, il est clairement plus efficace que la trace calculée précédemment soit simplement copiée sur le nœud courant. De cette manière, le travail est donc directement terminé pour cette partie de l'espace de recherche et nous pouvons donc passer directement à la suite. Ceci permet donc de gagner du temps sur la recherche exhaustive des solutions, ainsi que de l'espace pour ce qui est de la taille de la formule compilée puisqu'il suffit de faire une référence à une même branche.

Il existe plusieurs algorithmes de comptage de modèles basés sur ces méthodes. Les approches principales faisant état de l'art dans le comptage de modèles direct sont *Cachet* [SBB+04b] et *sharpSAT* [Thu06]. Les compteurs de modèles exacts sont très efficaces et ce même pour les problèmes avec un grand nombre de solutions. Cependant ils dépendent fortement de la répartitions des solutions dans l'arbre de recherche. En effet, il est évident qu'ils sont très efficaces lorsque les solutions sont concentrées dans un même endroit de cet espace. Lorsqu'elles sont éparpillées de manière plus éparse les compteurs directs sont beaucoup moins performants. Ceci est souvent le cas pour la plupart des instances de problèmes réels. Afin de répondre rapidement aux requêtes, dans certains cas il est possible pour certains problèmes qu'une estimation correcte du nombre de modèles soit suffisante.

2.3.2 Le comptage approché de modèles

Afin de gagner du temps, ou de répondre dans un temps imparti, il est aussi possible d'obtenir une estimation du nombre de modèles d'une formule propositionnelle. Il est parfois inutile pour certaines applications de connaître le nombre exact de modèles. Certaines applications n'ont pas besoin de distinguer qu'une formule possède 10^{51} ou $10^{51} + 9$ modèles. Une estimation grossière est suffisante pour résoudre ces problèmes. Dans ce cas, une approche qui fournie une estimation correcte, dans un temps respectable, et suffisamment fiable pour l'utilisateur est préférée. Lors d'une estimation deux informations sont à considérer : la qualité de l'estimation et la confiance en l'estimation. La qualité de l'estimation représente sa capacité à se rapprocher le plus possible du nombre de modèles exact de la formule. La confiance en l'estimation indique le pourcentage de confiance que nous pouvons avoir en cette estima-

tion. Elle peut être représentée sous plusieurs formes. Elle peut être représenté par une garantie sous forme de probabilité ou sur un intervalle ϵ, indiquant que l'estimation est correcte en tant que borne supérieure ou inférieure. Un compteur de modèle exact est donc d'une qualité et d'une confiance de 100%. L'idée des compteurs approchés est de jouer sur ces deux paramètres afin de réduire le temps de comptage, de manière à rester le plus précis possible. Il existe principalement deux méthodes, les estimations sans garanties et les estimations offrant des bornes supérieure et inférieure avec une certaines garantie du nombre de modèles. Ces méthodes sont :

— Les estimations sans garantie : ce sont des approches rapides et qui résistent assez bien à la montée en charge de la taille des problèmes. Cependant elles ne fournissent aucune garantie sur la confiance en l'estimation. Les principales approches de ce types sont basées sur l'échantillonnage. L'idée est que si nous sommes capable de partitionner une formule de façon quasi uniforme depuis l'ensemble de ses solutions, nous pouvons alors calculer une bonne estimation du nombre de modèles de la formule [WES04, WS05]. Par exemple, *ApproxCount* [WS05] s'appuie sur des méthodes de recherche locale et de calcul d'échantillons via une méthode de type Monte-Carlo pour calculer une approximation du nombre de modèles ;

— Les estimations bornées avec garanties : ce sont des approches rapides qui fournissent une garantie sur la borne (supérieure ou inférieure) donnée. Une approche simple et efficace d'obtenir une borne inférieure du nombre de modèles est d'exécuter une approche de comptage basé sur DPLL pendant un temps donné. Le comptage de modèles étant calculé de façon incrémentale, si l'algorithme dépasse le temps escompté il peut toujours donner une borne inférieure correcte du nombre de modèles de la formule propositionnelle. Cette borne correspondant au nombre de modèles associés à l'arbre de recherche partiel que l'approche a pu explorer pendant le temps imparti. Cette méthode peut aussi être utilisée pour définir une borne supérieure. En effet, il suffit de retourner $2^n - \mathcal{T}$, avec n le nombre de variables de la formule propositionnelle et $\mathcal{T}$ le nombre de contre modèles calculés dans l'espace de recherche exploré. Cependant, cette approche est en général peut convaincante en pratique car le nombre de modèles des formules est souvent beaucoup plus petit que la taille de l'espace de recherche. Cette approche mène donc à des bornes supérieures très grande, ce qui n'apporte que trop peu d'informations sur la formule propositionnelle.

Beaucoup d'autres approches ont été étudiées pour estimer de bonnes bornes du nombre de modèles. L'approche *SampleCount* [GHSS07] se base sur une estimation par échantillonnage comme *ApproxCount*. Il est montré qu'en modifiant la stratégie d'échantillonnage il est possible de garantir une grande confiance, donné sous forme de probabilité, sur l'estimation calculée sans aucune nécessité sur l'uniformité de l'échantillonnage. Cette approche ne donne qu'une borne inférieure du nombre de modèle de la formule. L'avantage de cette approche est que la qualité de l'uniformité de l'échantillonnage ne va affecter que la qualité de l'estimation de la borne inférieure et non sa confiance.

2.3.3 Le comptage de modèles via compilation de connaissances

Une autre approche pour le comptage exact de modèles est de compiler la formule `CNF` vers un autre langage permettant de compter plus facilement le nombre de modèles via la formule obtenue. L'idée principale de la compilation de connaissances est de passer beaucoup de temps sur la partie difficile, qui est la traduction de la formule écrite dans un langage vers un autre langage, et de compenser ce temps grâce à l'exécution de nombreuses requêtes qui seront beaucoup plus simples à résoudre avec une formule dans ce nouveau langage.

Comme nous le verrons dans le chapitre suivant, plusieurs langages permettent de compter le nombre de modèles d'une formule en temps polynomial. Cependant, l'un d'entre eux est utilisé en pratique par quelques compilateurs, il s'agit des *deterministic Decomposable Negation Normal Form* (`d-DNNF`).

Deux principaux compilateurs ont été développés pour compiler une formule CNF vers les d−DNNF, c2d [Dar04] et *Dsharp* [MMBH12]. Ils sont basés sur la trace d'un solver DPLL et se sont montrés performants pour différentes familles d'instances. Ils sont même parfois plus efficaces que les compteurs exacts directs tels que *Cachet*, et ceux parfois même dès la première requête posée. Cette approche est clairement plus efficace pour les applications faisant plusieurs fois appel au comptage de modèles puisque l'approche compilée ne calcule la partie difficile qu'une seule fois, tandis que les compteurs directs recalculent l'ensemble du problème à chaque requête.

Chapitre 3

La compilation de connaissances

Sommaire

3.1 Propriétés des langages cibles pour la compilation de formules propositionnelles . 44
3.1.1 Efficacité temporelle 44
3.1.2 Liens entre requêtes et transformations 47
3.1.3 Efficacité spatiale 48
3.2 Langages cibles pour la logique propositionnelle 49
3.2.1 NNF et ses sous-ensembles 49
3.2.2 Les diagrammes de décision binaire 53
3.2.3 Les arbres de décision (DT) 57
3.2.4 Le langage AFF des formules affines 60
3.3 Résultats de la carte de compilation 65

Dans ce chapitre, nous présentons le cadre dans lequel la contribution de la thèse s'inscrit, à savoir la compilation de connaissances. Nous commençons par présenter le principe et l'objectif de la compilation de connaissances. Nous nous focalisons ensuite sur les critères à prendre en compte pour comparer les différents langages cibles possibles pour la compilation et dresser ainsi une carte de compilation pour les formules propositionnelles. Il s'agit d'une part de l'efficacité temporelle du langage, c'est-à-dire l'existence ou la non-existence (prouvée ou vraisemblable sous les hypothèses usuelles de la théorie de la complexité) d'algorithmes en temps polynomial pour réaliser différents traitements. Il s'agit plus précisément de répondre à telle ou telle requête et effectuer telle ou telle transformation sur des représentations issue du langage cible considéré. Il s'agit aussi de l'efficacité spatiale (relative) du langage, c'est-à-dire la possibilité et/ou l'impossibilité (là encore, prouvée ou vraisemblable sous les hypothèses usuelles de la théorie de la complexité) de traduire les représentations d'un langage en représentations d'un autre en espace polynomial. Nous rappelons ensuite les définitions de plusieurs langages cibles introduits jusqu'ici, en particulier ceux de la famille d−DNNF mais aussi les formules affines AFF et les disjonctions de formules affines AFF[∨] et précisons leur efficacité temporelle et spatiale telle qu'elles ont été identifiées dans la carte de compilation pour les formules propositionnelles.

En intelligence artificielle, l'approche à base de connaissances est basée sur l'idée qu'un agent enregistre ce qu'il connait dans une base de connaissances, de façon déclarative. Cette base de connaissances est souvent représentée dans un formalisme logique. L'intérêt d'utiliser une telle représentation déclarative est que l'information stockée existe indépendamment des traitements pouvant être mis en œuvre pour l'exploiter. Parmi eux figure la requête d'interrogation, qui consiste à déterminer si une information donnée est conséquence ou non de la base de connaissances. Celle-ci peut être calculatoirement difficile,

c'est le cas en particulier quand la base de connaissances prend la forme d'une formule propositionnelle ou d'un réseau de contraintes (le problème d'interrogation est coNP-complet dans cette situation).

Pour pallier cette complexité de calcul, la compilation de connaissances est une approche possible. Elle consiste à traduire la base de connaissances dans un autre formalisme afin de répondre plus efficacement à différents traitements, par exemple à plusieurs requêtes d'interrogation. Ainsi, en logique propositionnelle, si la base de connaissances considérée est décrite par l'ensemble de ses solutions, il est alors facile de déterminer si une requête donnée (sous forme d'une formule construite sur les mêmes variables que la base) en constitue une conséquence logique. L'ensemble des représentations des formules propositionnelles par leurs ensembles de solutions (ou modèles) est un langage cible (fort peu compact) pour la compilation, que l'on note MODS.

L'apport de la compilation du point de vue computationnel vient du fait que les différents traitements à réaliser sur les connaissances portent souvent sur deux parties distinctes : une partie « fixe » (ou du moins qui n'évolue pas vite), par exemple, la base de connaissances et une partie « variable », par exemple les requêtes d'interrogation à prendre en compte. L'idée est simplement de pré-traiter (compiler) la partie « fixe » une bonne fois pour toutes, pour ensuite, quand la partie « variable » est connue, réaliser plus efficacement le traitement voulu. La phase de compilation est appelée la phase *off-line*, elle est effectuée une seule fois et doit préserver les informations utiles à la résolution du problème traité (par exemple, l'ensemble des solutions). Puis les traitements sont effectivement réalisés, en utilisant la forme compilée de la base de connaissances en lieu et place de la base originale. Cette phase est appelée la phase *on-line* (ou phase de requêtes).

Le principe de la compilation de connaissances est de consacrer un effort calculatoire éventuellement important pendant la phase *off-line* si cet effort peut être compensé, c'est-à-dire s'il est possible d'amortir le temps passé durant cette phase en permettant de réaliser beaucoup plus vite les traitements (parfois nombreux) à réaliser lors de la phase *on-line*. La phase *on-line* est ainsi rendue plus simple du point de vue du calcul grâce à la traduction réalisée lors de la phase *off-line*.

Exemple 29 *Soit la formule propositionnelle sous forme* CNF $\Sigma = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$. *Sous cette forme, il est en général difficile de déterminer si une formule possède un modèle (c'est le problème SAT) ou d'en compter le nombre de modèles (c'est le problème #SAT). Nous représentons maintenant* Σ *sous la forme de l'ensemble de ses modèles :*

x_1	x_2	x_3
1	0	0
1	0	1
0	1	0
0	0	1

Dans ce tableau (que l'on peut voir comme une représentation de Σ *dans le langage* MODS*, un* 1 *indique que la valeur de vérité de la variable est à vrai, un* 0 *qu'elle est à faux. Sous cette forme, nous pouvons facilement conclure que* Σ *est cohérente. Le nombre de modèles de* Σ *est aussi facilement calculable puisqu'il suffit de compter le nombre de lignes du tableau représentant* Σ*.*

Plus formellement, la compilation de connaissances porte sur des traitements f à réaliser (f est une certaine fonction), pour lesquels chaque entrée possible prend la forme d'un couple (Σ, α_i) :

— Σ est la partie fixe de l'instance (typiquement, commune à plusieurs parties variables) ;

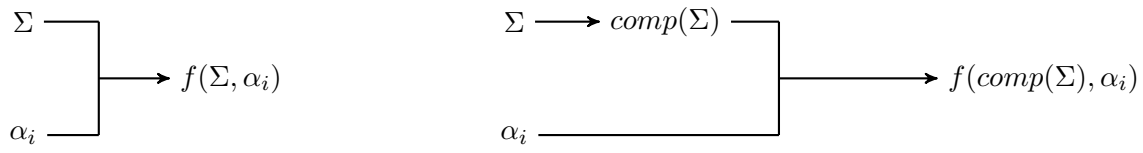
— α_i est la partie variable de l'instance.

Par exemple, si f modélise la requête d'interrogation d'une base de connaissances, alors Σ représente la base de connaissances et chaque α_i est une requête à traiter.

La figure 3.1a schématise l'évaluation directe de f sur l'entrée (Σ, α_i). $f(\Sigma, \alpha_i)$ est le résultat du traitement. La figure 3.1b schématise l'évaluation de f sur l'entrée (Σ, α_i) après compilation de Σ. La partie fixe Σ est d'abord traduite en une forme compilée $comp(\Sigma)$ et l'évaluation de f est réalisée sur l'entrée $(comp(\Sigma), \alpha_i)$. Les résultats retournés par $f(\Sigma, \alpha_i)$ et $f(comp(\Sigma), \alpha_i)$ sont les mêmes quand Σ et $comp(\Sigma)$ sont équivalents.

Pour que l'approche par compilation conduise à des bénéfices calculatoires au niveau des instances traitées, il faut que le temps mis pour évaluer l'ensemble des $f(comp(\Sigma), \alpha_i)$ quand α_i varie soit au final plus petit que le temps mis pour évaluer l'ensemble des $f(\Sigma, \alpha_i)$ quand α_i varie. En pratique, cela pourra être le cas quand le calcul de $f(\Sigma, \alpha_i)$ est NP-difficile quand Σ n'est pas sous forme compilée alors que le calcul de $f(comp(\Sigma), \alpha_i)$ peut être réalisé en temps polynomial. Toutefois, le fait de faire baisser la complexité de l'évaluation de f lors de la phase en ligne quand on a compilé Σ ne suffit pas à garantir que l'approche par compilation a de l'intérêt au niveau du problème considéré (i.e., pour ce qui concerne le calcul de f en toute généralité, sans préciser les instances considérées). En effet, la complexité d'un algorithme s'exprime en toute généralité comme une fonction de la taille de son entrée. Si le meilleur algorithme connu pour calculer f selon l'approche directe (donc sans compilation) a une complexité temporelle en $\Omega(2^n)$ et que le meilleur algorithme connu pour calculer f selon l'approche par compilation a une complexité temporelle en $O(n)$, on ne peut pas forcément conclure à l'intérêt de cette dernière approche. Ainsi, si la taille $|comp(\Sigma)|$ de la forme compilée de $comp(\Sigma)$ est exponentielle en la taille de Σ, le temps mis en pratique pour calculer $f(comp(\Sigma), \alpha_i)$ (avec un algorithme en temps linéaire) peut très bien excéder celui mis pour calculer $f(\Sigma, \alpha_i)$ (avec un algorithme pourtant en temps exponentiel). Pour cette raison, compiler une base de connaissances donnée par une formule propositionnelle vers le langage cible MODS pour pouvoir l'interroger plus efficacement n'est, en général, pas une bonne idée (le nombre de modèles d'une formule propositionnelle étant souvent exponentiel dans la taille de sa représentation dans des langages usuels, comme le langage CNF).

En théorie, beaucoup de traitements intéressants mais calculatoirement difficiles f ne sont pas compilables dans P, i.e., on ne connait pas de fonction de compilation $comp$ qui garantirait d'une part que $|comp(\Sigma)|$ soit toujours polynomiale dans $|\Sigma|$ pour toute entrée Σ possible et d'autre part pour lesquels l'évaluation de f pourrait être réalisée en temps polynomial à partir de la forme compilée alors qu'elle est NP-difficile sinon. Pour divers f (comme l'interrogation, même quand les requêtes d'interrogation se limitent à des formules « simples » comme des clauses), on peut même montrer que de telles fonctions $comp$ ne peuvent pas exister sauf si NP $\subseteq$ P/poly (une hypothèse jugée peu vraisemblable en théorie de la complexité) [CD98, CDLS02]. Cela n'enlève toutefois rien à l'apport que la compilation peut fournir en pratique : ainsi, même lorsqu'il ne peut être assuré que $|comp(\Sigma)|$ soit toujours polynomiale dans $|\Sigma|$, si pour Σ donné, $|comp(\Sigma)|$ reste « relativement petit », le temps mis à compiler Σ pourra être compensé. En outre, on pourra souvent majorer finement le temps nécessaire pour calculer $f(comp(\Sigma), \alpha_i)$ (et assurer ainsi des garanties de temps de réponse s'il est réduit) alors que le temps nécessaire pour calculer $f(\Sigma, \alpha_i)$ sera très élevé dans le pire des cas. Pour poursuivre l'exemple ci-dessus, si le nombre de modèles de Σ est réduit, compiler Σ dans le langage MODS pour pouvoir interroger Σ plus efficacement a parfaitement du sens : le temps nécessaire pour répondre à une requête d'interrogation est linéaire dans la taille de sa représentation en MODS et la taille de la requête alors qu'aucun algorithme en temps polynomial n'existe pour cela si Σ est représentée dans le langage CNF (et qu'il n'en existe pas, sauf si P = NP). De telles garanties de temps de réponse sont attendues dans diverses applications, en particulier quand il s'agit de fournir des réponses aux sollicitations d'un utilisateur qui interagit à distance avec un système informatique, par exemple via le Web.



(a) Approche directe (b) Approche compilée

3.1 Propriétés des langages cibles pour la compilation de formules propositionnelles

Dans cette section, nous présentons les critères qui ont été proposés pour évaluer les langages cibles pour la compilation de formules propositionnelles et, sur cette base, de choisir un « bon » langage cible selon l'application traitée. Il s'agit d'une part de critères portant sur l'efficacité temporelle du langage cible considéré $\mathcal{L}$, c'est-à-dire l'existence ou la non-existence (prouvée ou vraisemblable sous les hypothèses usuelles de la théorie de la complexité) d'algorithmes en temps polynomial pour réaliser différents traitements. Ces traitements sont classés en requêtes - les traitements qui extraient de l'information de la forme compilée, par exemple son nombre de solutions - et en transformations - les traitements qui permettent de produire ou de modifier une représentation sous forme compilée. Chaque traitement est également vu comme une propriété que le langage $\mathcal{L}$ vérifie ou ne vérifie pas : la propriété considérée est vérifiée par $\mathcal{L}$ quand un algorithme en temps polynomial pour réaliser le traitement peut être mis en évidence et la propriété n'est pas vérifiée sinon. Il s'agit aussi de l'efficacité spatiale (relative) du langage $\mathcal{L}$ par rapport à d'autres langages cibles possibles $\mathcal{L}'$, c'est-à-dire la possibilité et/ou l'impossibilité (avérée ou vraisemblable sous les hypothèses usuelles de la théorie de la complexité) de traduire les représentations de $\mathcal{L}$ en représentations de $\mathcal{L}'$ équivalentes et de tailles polynomiales, et vice-versa.

La carte de compilation initiée dans [DM02] (puis complétée et étendue dans d'autres articles), consiste en l'évaluation (multi-critère, selon les critères évoqués ci-dessus et décrits ci-après) d'un ensemble de langages cibles possibles. Cette carte constitue un support méthodologique pour choisir un langage cible. Lorsque l'application visée requiert la réalisation d'un ensemble de requêtes et de transformations données et qu'il s'agit d'améliorer l'efficacité (en temps) de la mise en œuvre de celles-ci via un pré-traitement, une politique de choix possible consiste à retenir en priorité un langage cible qui offre toutes (ou un ensemble maximal) des propriétés correspondantes et parmi eux, préférer l'un des plus efficaces spatialement.

3.1.1 Efficacité temporelle

Requêtes

Dans cette section, nous présentons les huit requêtes de base considérées dans la carte de compilation [DM02], à savoir le test de cohérence, le test de validité, le test d'implication, le test d'impliquant, le test d'équivalence, le test d'implication générale, le test de comptage du nombre de modèles et l'énumération des modèles d'une base de connaissances.

Soit $\mathcal{L}$ un langage cible de compilation. Les requêtes listées ci-dessus sont définies formellement de la manière suivante :

Cohérence (CO) $\mathcal{L}$ satisfait la propriété du test de cohérence **CO** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne *vrai* si Σ est cohérente, *faux* sinon.

Exemple 30 *Le langage* CNF *ne vérifie pas la propriété du test de cohérence (sauf si* P=NP*).*

Validité (VA) $\mathcal{L}$ satisfait la propriété du test de validité **VA** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne *vrai* si Σ est valide, *faux* sinon.

Exemple 31 *Le langage* CNF *vérifie la propriété du test de validité. En effet, une conjonction de clauses est valide si et seulement si chaque clause de la conjonction est valide, c'est-à-dire si elle contient* $\top$ *ou deux littéraux complémentaires. Ceci se teste efficacement.*

Implication clausale (CE) $\mathcal{L}$ vérifie la propriété du test d'implication clausale **CE** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et toute clause γ retourne *vrai* si $\Sigma \models \gamma$ et *faux* sinon.

Exemple 32 *Le langage* CNF *ne vérifie pas la propriété du test d'implication clausale (sauf si* P=NP*).*

Impliquant (IM) $\mathcal{L}$ vérifie la propriété du test d'impliquant **IM** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout terme cohérent γ retourne *vrai* si $\gamma \models \Sigma$ et *faux* sinon.

Exemple 33 *Le langage* CNF *vérifie la propriété du test d'impliquant. En effet, un terme cohérent implique une clause si et seulement si la clause est valide ou le terme contient un littéral de la clause, ce qui peut être testé efficacement.*

Équivalence (EQ) $\mathcal{L}$ satisfait la propriété du test d'équivalence **EQ** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ_i et Σ_j de $\mathcal{L}$ retourne *vrai* si $\Sigma_i \equiv \Sigma_j$, *faux* sinon.

Implication générale (SE) $\mathcal{L}$ satisfait la propriété du test d'implication générale **SE** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ_i et Σ_j de $\mathcal{L}$ retourne *vrai* si $\Sigma_i \models \Sigma_j$ et *faux* sinon.

Comptage de modèles (CT) $\mathcal{L}$ satisfait la propriété **CT** de comptage de modèles si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne un entier naturel qui est le nombre de modèles de Σ.

Énumération de modèles (ME) $\mathcal{L}$ satisfait la propriété **ME** d'énumération de modèles [1] si et seulement s'il existe un polynôme p et un algorithme qui retourne tous les modèles d'une formule Σ de $\mathcal{L}$ en temps $p(n, m)$ où n est la taille de Σ et m son nombre de modèles (sur les variables de Σ).

Exemple 34 *Le langage* CNF *ne vérifie pas les tests d'équivalence, de l'implication générale, du comptage de modèles ou l'énumération de modèles (sauf si* P=NP*).*

Le tableau 3.2 résume les requêtes considérées, ainsi que leurs acronymes.

1. On pourrait renforcer cette propriété et exiger que l'énumération soit à délai polynomial.

Acronyme	Requête
CO	Test de la cohérence
VA	Test de la validité
CE	Test de l'implication clausale
IM	Test d'impliquant
EQ	Test de l'équivalence
SE	Test d'implication générale
CT	Comptage de modèles
ME	Eńumération de modèles

FIGURE 3.2 – Acronyme des requêtes

Transformations

Dans cette section, nous présentons la liste des transformations considérées dans [DM02]. Il s'agit du conditionnement, de l'oubli (possiblement borné) et des clôtures (possiblement bornées) par les connecteurs logiques $\wedge$ et $\vee$, ainsi que par la négation $\neg$.

Conditionnement (CD) $\mathcal{L}$ vérifie la propriété de conditionnement **CD** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout terme cohérent γ retourne une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma|\gamma$ (logiquement équivalent à la formule $\Sigma \wedge \gamma$).

Exemple 35 *Le langage* CNF *vérifie la propriété de conditionnement. Il suffit de remplacer dans la formule par $\top$ ou $\bot$ les variables conditionnées selon leur polarité dans le terme avec lequel nous conditionnons. Ce remplacement s'effectue en temps linéaire dans la taille de la formule et préserve la forme normale conjonctive de celle-ci.*

Oubli (FO) $\mathcal{L}$ vérifie la propriété de l'oubli **FO** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout sous-ensemble de variables X retourne une formule de $\mathcal{L}$ logiquement équivalente à $\exists X.\Sigma$.

L'oubli des variables de X dans Σ consiste à supprimer toute occurrence de X dans Σ tout en maintenant les informations ne concernant pas $\mathcal{X}$ que Σ capture. Formellement, il s'agit de la formule notée $\exists X.\Sigma$ qui est la conséquence logique la plus générale de Σ qui soit indépendante de X (i.e., elle équivaut à une formule dans laquelle aucune variable de X n'apparaît). $\exists X.\Sigma$ est unique à l'équivalence logique près.

Oubli d'une variable (SFO) $\mathcal{L}$ vérifie la propriété de l'oubli d'une variable **SFO** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout singleton X retourne une formule de $\mathcal{L}$ logiquement équivalente à $\exists X.\Sigma$.

Exemple 36 *Le langage* CNF *ne vérifie pas la propriété de l'oubli (sauf si* P=NP*), mais vérifie celle de l'oubli d'une variable.*

Conjonction ($\wedge$ C) $\mathcal{L}$ vérifie la propriété de la conjonction $\wedge$**C** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à tout ensemble fini de formules $\{\Sigma_1, \ldots, \Sigma_n\}$ de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_1 \wedge \cdots \wedge \Sigma_n$.

Conjonction bornée ($\wedge$BC) $\mathcal{L}$ vérifie la propriété de la conjonction bornée $\wedge$**BC** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à toute paire de formules Σ_i, Σ_j de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_i \wedge \Sigma_j$.

Exemple 37 *Le langage* CNF *vérifie la propriété de la conjonction (donc celle de la conjonction bornée).*

Disjonction ($\vee$C) $\mathcal{L}$ vérifie la propriété de la disjonction $\vee$**C** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à tout ensemble fini de formules $\{\Sigma_1, \ldots, \Sigma_n\}$ de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_1 \vee \cdots \vee \Sigma_n$.

Disjonction bornée ($\vee$BC) $\mathcal{L}$ vérifie la propriété de la disjonction bornée $\vee$**BC** si et seulement s'il existe un algorithme en temps polynomial qui permet d'associer à toute paire de formules Σ_i, Σ_j de $\mathcal{L}$ une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_i \vee \Sigma_j$.

Exemple 38 *Le langage* CNF *ne vérifie par la propriété de la disjonction (sauf si* P=NP*), mais vérifie celle de la disjonction bornée.*

Négation ($\neg$C) $\mathcal{L}$ vérifie la propriété de la négation $\neg$**C** si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ retourne une formule de $\mathcal{L}$ logiquement équivalente à $\neg\Sigma$.

Exemple 39 *Le langage* CNF *ne vérifie pas la propriété de la négation.*

Le tableau 3.3 résume les transformations considérées, ainsi que leurs acronymes.

Acronyme	Transformation
CD	Conditionnement
FO	L'oubli
SFO	L'oubli d'un singleton
$\neg$**C**	La négation
$\wedge$**C**	La conjonction
$\wedge$**BC**	La conjonction bornée
$\vee$**C**	La disjonction
$\vee$**BC**	La disjonction bornée

FIGURE 3.3 – Acronyme des transformations

3.1.2 Liens entre requêtes et transformations

Des liens formels entre requêtes et transformations (vues comme des propriétés des langages cibles) existent et ont été identifiés dans [DM02]. Ainsi, lorsque certaines propriétés sont satisfaites par un langage, d'autres propriétés sont automatiquement offertes par celui-ci.

Par exemple, un langage qui satisfait le test d'équivalence (**EQ**) vérifie aussi les tests de validité (**VA**) et de cohérence (**CO**), pour autant qu'il permette de représenter les formules valides et les formules contradictoires (ce qui est le cas si le langage est complet). En effet, une formule est valide si et seulement si elle est équivalente à la formule « $vrai$ » et une formule est incohérente si et seulement si elle

est équivalente à « $faux$ ». De même, un langage qui satisfait le test d'implication générale (**SE**) vérifie aussi l'équivalence (**EQ**), la validité (**VA**) et la cohérence (**CO**), puisque deux formules Σ_1 et Σ_2 sont logiquement équivalentes si et seulement si $\Sigma_1 \models \Sigma_2$ et $\Sigma_2 \models \Sigma_1$. Un langage qui vérifie le test d'implication générale vérifie donc le test d'équivalence. Par conséquent, il satisfait aussi les tests de validité et de cohérence.

L'une des principales application du conditionnement vient du fait que pour formule Σ et terme cohérent γ, on a

$$\Sigma \wedge \gamma \equiv (\Sigma \mid \gamma) \wedge \gamma.$$

Soit Σ une formule de $\mathcal{L}$ et δ une clause (que l'on peut supposer non tautologique : si δ est valide – ce qui peut être décidé efficacement, on a $\Sigma \models \delta$ quel que soit Σ). Déterminer si $\Sigma \models \delta$ revient à déterminer la cohérence de la formule $\Sigma \wedge \neg\delta$, donc celle de $(\Sigma \mid \neg\delta) \wedge \neg\delta$. Et comme $\neg\delta$ équivaut à un terme cohérent (quand δ n'est pas valide) et ne partage aucune variable avec $\Sigma \mid \neg\delta$ (par construction), cela revient à déterminer la cohérence de $\Sigma \mid \neg\delta$. Comme on peut passer en temps linéaire de $\neg\delta$ à un terme γ (cohérent) équivalent, il suffit de conditionner Σ par le terme γ et de tester la cohérence de la formule obtenue pour déterminer si $\Sigma \models \delta$. Cela conduit donc à un algorithme en temps polynomial pour de résoudre le test d'implication clausale (**CE**). Donc si un langage $\mathcal{L}$ permet le conditionnement (**CD**) et le test de cohérence (**CO**) en temps polynomial, alors il vérifie aussi le test de l'implication clausale (**CE**) en temps polynomial.

Si un langage satisfait le test de cohérence (**CO**) et offre la négation ($\neg$**C**) et la conjonction ($\wedge$**C**), alors il satisfait le test d'implication générale (**SE**). En effet, tester si deux formules Σ_1 et Σ_2 vérifient $\Sigma_1 \models \Sigma_2$ revient à tester si la formule $\Sigma_1 \wedge \neg\Sigma_2$ est incohérente.

De même, si un langage satisfait le test de validité (**VA**) et offre la négation ($\neg$**C**) et la disjonction ($\vee$**C**), alors il offre le test d'implication générale (**SE**). En effet, tester si deux formules Σ_1 et Σ_2 vérifient $\Sigma_1 \models \Sigma_2$ revient à tester si la formule $\neg\Sigma_1 \vee \Sigma_2$ est valide.

Un langage qui satisfait le test d'impliquant (**IM**) satisfait aussi le test de validité (**VA**). En effet, s'il est possible de tester en temps polynomial si $\gamma \models \Sigma$, avec Σ une formule et γ un terme quelconque, alors c'est vrai en particulier pour $\gamma = \top$ et dans ce cas, on a $\gamma \models \Sigma$ si et seulement si Σ est valide. De façon similaire, un langage qui satisfait le test d'implication clausale (**CE**) satisfait aussi le test de cohérence (**CO**). Il suffit cette fois de considérer la requête $\delta = \bot$ (la clause vide) : on a $\Sigma \models \delta$ si et seulement si Σ est contradictoire lorsque δ est contradictoire.

Enfin, tout langage qui satisfait la propriété du comptage de modèles (**CT**) satsifait aussi les tests de validité (**VA**) et de cohérence (**CO**). En effet, tester la validité (respectivement, la cohérence) d'une formule Σ revient à déterminer si elle possède $2^{|Var(\Sigma)|}$ modèles (respectivement, au moins un modèle).

3.1.3 Efficacité spatiale

Nous terminons cette section en définissant l'efficacité spatiale (relative) d'un langage [GKPS95]. Comme on l'a évoqué, elle indique la capacité (relative) d'un langage à représenter succinctement de l'information en comparaison à une autre. Cette efficacité se définit comme suit :

Définition 32 (Efficacité spatiale) *Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages propositionnels. Le langage $\mathcal{L}_1$ est au moins aussi succinct que $\mathcal{L}_2$, noté $\mathcal{L}_1 \leq_s \mathcal{L}_2$, si et seulement si il existe un polynôme p tel que pour tout $\alpha \in \mathcal{L}_2$, il existe $\beta \in \mathcal{L}_1$ qui est logiquement équivalente à α et telle que $\mid \beta \mid \leq_s p(\mid \alpha \mid)$.*

Cette notion d'efficacité spatiale n'impose pas qu'il existe une traduction en temps polynomial de $\mathcal{L}_2$ vers $\mathcal{L}_1$, mais (ce qui est moins restrictif) qu'une traduction dont le résultat exprimé dans $\mathcal{L}_1$ est de taille polynomiale (pour un polynôme fixé) dans la taille de sa représentation dans $\mathcal{L}_2$. En particulier, aucune hypothèse n'a besoin d'être faite sur le temps nécessaire à la traduction.

Il est aisé de prouver que la relation $\leq_s$ est réflexive et transitive, elle constitue donc un pré-ordre. Nous pouvons ainsi définir sa partie stricte, i.e., la relation $<_s$ telle que $\mathcal{L}_1 <_s \mathcal{L}_2$ si et seulement si $\mathcal{L}_1 \leq_s \mathcal{L}_2$ et $\mathcal{L}_2 \not\leq_s \mathcal{L}_1$. $\mathcal{L}_1 <_s \mathcal{L}_2$ signifie que $\mathcal{L}_1$ est strictement plus succinct que $\mathcal{L}_2$.

3.2 Langages cibles pour la logique propositionnelle

Dans cette section, nous présentons quelques langages cibles de compilation développés jusqu'ici en la logique propositionnelle. Pour plus de détails, le lecteur intéressé peut se référer, en particulier, à [DM02, FM08b, FM08a].

3.2.1 NNF et ses sous-ensembles

Nous nous focalisons surtout sur des sous-ensembles du langage NNF des circuits (ou graphes) dans lesquels les portes sont de type $\wedge$ ou $\vee$ et les négations ne portent que sur les entrées du circuit. Ces dernières sont des variables propositionnelles ou des constantes booléennes ($\top$ ou $\bot$).

Définition 33 (NNF) *Un formule NNF est un graphe fini, non vide, orienté et sans circuit (DAG), ayant une racine unique et où chaque feuille est étiquetée par $\top$, $\bot$ ou une variable propositionnelle x possiblement niée ($\neg x$). Chaque nœud interne est étiqueté par une conjonction $\wedge$ ou une disjonction $\vee$ et peut avoir un nombre quelconque (fini) de fils.*

La taille d'une formule Σ NNF, noté $|\Sigma|$, est le nombre d'arcs du graphe. Sa hauteur correspond au nombre maximum d'arcs entre la racine et l'une des feuilles du graphe. Pour alléger l'écriture et quand cela ne crée aucune ambiguité, on s'autorisera dans la suite à identifier tout nœud N d'une formule NNF avec la formule enracinée à ce nœud.

Bien que complet pour la logique propositionnelle, le langage NNF ne permet aucune requête intéressante du point de vue de la compilation (ni **CO**, ni a fortiori **CT**).

En revanche, l'ajout de restrictions permet de définir des langages sous-ensembles de NNF, offrant d'intéressantes propriétés pour la compilation. Ainsi, le langage DNNF est le sous-ensemble du langage NNF qui satisfait la propriété de décomposabilité.

Définition 34 (Décomposabilité) *Un nœud de conjonction $N_\wedge$ d'une formule NNF est décomposable si et seulement si ses fils $N_1, \ldots, N_k$ ne partagent aucune variable en commun. Formellement, $Var(N_i) \cap Var(N_j) = \varnothing$ pour tout $i \neq j$.*

Définition 35 (DNNF) *Une formule DNNF est une formule NNF dans laquelle chaque nœud de conjonction est décomposable.*

La figure 3.4 donne un exemple de formule DNNF.

Le langage DNNF se montre plus attractif que NNF puisqu'il satisfait trois des requêtes qui nous intéressent tout en préservant la transformation **CD**. Ainsi, DNNF satisfait le test de cohérence (**CO**), le test d'implication clausale (**CE**) et l'énumération des modèles (**ME**). Il ne permet cependant pas de compter le nombre de modèles d'une formule en temps polynomial sauf si P = NP.

L'ajout d'une restriction supplémentaire permet de rendre ce langage encore plus attractif. Le langage d-DNNF est le sous-ensemble du langage DNNF qui satisfait la propriété de déterminisme.

Définition 36 (Déterminisme) *Un nœud de disjonction $N_\vee$ d'une formule NNF est déterministe si et seulement si ses fils $N_1, \ldots, N_k$ sont deux à deux contradictoires. Formellement, $N_i \wedge N_j \models \bot$ pour tout $i \neq j$.*

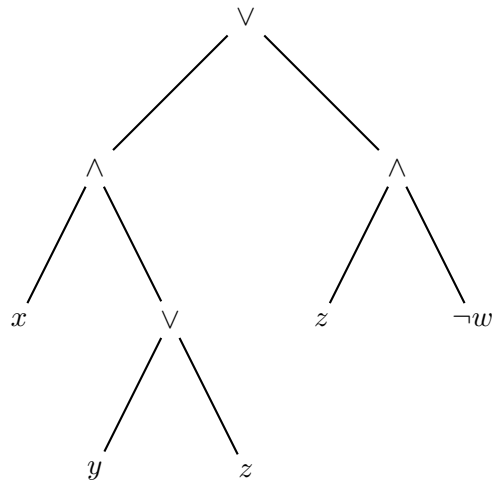


FIGURE 3.4 – Une formule DNNF

La figure 3.5 donne un exemple de formule d−DNNF.

Le langage d−DNNF est beaucoup plus attractif que DNNF du point de vue des requêtes offertes. En effet, ce langage satisfait toutes les propriétés correspondant aux requêtes proposées, sauf deux, le test d'implication générale (**SE**) et le test d'équivalence (**EQ**). En particulier, la propriété **CT** est offerte par d−DNNF. En effet, la propriété de la décomposabilité stipule que les fils de chaque nœud $\wedge$ n'ont aucune variable en commun. Par conséquent, le nombre de modèles d'un nœud $\wedge$ décomposable correspond au produit du nombre de modèles de ses fils.

Proposition 1 (Calcul du nombre de modèles d'un nœud $\wedge$ décomposable) *Soit $N_\wedge$ un nœud de conjonction décomposable et $N_1, \ldots, N_k$ ses fils. Le nombre de modèles de $N_\wedge$ vaut $\#N_\wedge = \#N_1 \times \cdots \times \#N_k$.*

Définition 37 (d−DNNF**)** *Une formule* d−DNNF *Σ est une formule* DNNF *dans laquelle chaque nœud de disjonction $\vee$ est déterministe. Chaque nœud interne d'une formule* d−DNNF *est donc soit un nœud de disjonction déterministe, soit un nœud de conjonction décomposable.*

La propriété de déterminisme du langage d−DNNF stipule que les fils de chaque nœud de disjonction d'une formule d−DNNF ne possèdent aucun modèle en commun. Ainsi, le nombre de modèles d'un nœud $\vee$ déterministe représente la somme du nombre de modèles de ses fils (à un facteur de normalisation près par fils, lié au fait que les fils peuvent être construits sur des ensembles de variables qui diffèrent). En particulier, aucune normalisation n'est nécessaire si les fils portent sur les mêmes variables :

Proposition 2 (Calcul du nombre de modèles d'un nœud $\vee$ déterministe) *Soit $N_\vee$ un nœud de disjonction déterministe et $N_1, \ldots, N_k$ ses fils. Si $Var(N_i) = Var(N_j)$ pour tout $i, j \in \{1, \ldots, k\}$, le nombre de modèles de $N_\vee$ vaut $\#N_\vee = \#N_1 + \cdots + \#N_k$.*

Le calcul du nombre de modèles d'une formule d−DNNF peut s'effectuer en un seul parcours récursif du graphe. Le principe est de partir de la racine, puis de parcourir l'ensemble de ses fils jusqu'aux feuilles, puis de remonter les valeurs d'un attribut synthétisé (le nombre de modèles de la formule enracinée à ce nœud) jusqu'à la racine. Si la feuille considérée est étiquetée par $\top$, la valeur retournée est 1. Si elle est étiquetée par $\bot$, la valeur retournée est 0. Sinon, la feuille représente un littéral et la valeur retournée est 1. Enfin, les nœuds $\wedge$ décomposables retournent le produit des valeurs de leur fils et les

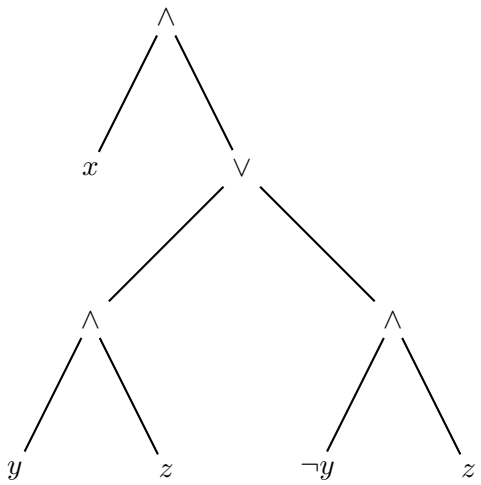


FIGURE 3.5 – Une formule d−DNNF

nœuds $\vee$ déterministes retournent la somme des valeurs de leurs fils (après produit par un facteur de cadrage défini à partir du nombre de variables figurant dans les frères du fils traité mais pas dans le fils lui-même).

Aucun facteur de cadrage n'est à prendre en compte lorsque les fils $N_1, \ldots, N_k$ de chaque nœud de disjonction déterministe de la formule d−DNNF considérée Σ vérifient la propriété d'uniformité (en anglais, *smoothness*) :

Définition 38 (Uniformité) *Un nœud de disjonction $N_\vee$ d'une formule* NNF *est uniforme si et seulement si ses fils $N_1, \ldots, N_k$ portent sur les mêmes variables. Formellement, $Var(N_i) = Var(N_j)$ pour tout i, j.*

On dit alors que la formule Σ est une formule d−DNNF uniforme. Il est facile d'associer à toute formule d−DNNF Σ en temps linéaire une formule d−DNNF uniforme équivalente [DM02].

L'algorithme 4 décrit comment on peut calculer le nombre de modèles d'une formule d−DNNF uniforme enracinée en N. Si N est une feuille étiquetée par $\bot$, la valeur 0 est retournée. Si N est une feuille autre que $\bot$, alors la valeur 1 est retournée. Si N est un nœud $\wedge$ décomposable, alors l'algorithme est appelé récursivement pour chaque fils de N, et le résultat du produit des valeurs obtenues pour chaque fils est retourné. Enfin, siN est un nœud $\vee$ déterministe et uniforme, alors l'algorithme est appelé récursivement pour chaque fils de N, et le résultat de la somme des valeurs obtenues pour chaque fils est retourné. Les valeurs sont remontées ainsi de suite jusqu'à la racine de la formule considérée.

La figure 3.6 indique comment on peut calculer le nombre de modèles de la formule d−DNNF uniforme de la figure 3.5 en effectuant un tel calcul d'attribut. Chaque nœud retourne sa valeur, en réalisant l'opération indiquée. La valeur retournée à la racine représente le nombre de modèles de la formule d−DNNF. Cette formule possède donc 2 modèles.

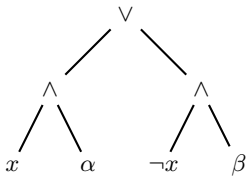
Un autre sous-ensemble intéressant de d−DNNF, le langage Decision−DNNF, est défini en imposant une restriction supplémentaire sur les nœud $\vee$ des formules. Ce langage est basé sur la notion de nœud de décision, où un nœud de décision est un nœud $\vee$ déterministe particulier.

Définition 39 (Nœud de décision) *Un nœud de disjonction $N_\vee$ d'une formule* NNF *est un nœud de décision s'il a la forme suivante :*

Algorithme 4 : sharpdDnnf(N)

entrée : La racine N d'une formule `d-DNNF`
sortie : Le nombre de modèles de N (sur $Var(N)$)

```
si leaf(N) alors
    si N = ⊥ alors
        return 0
    sinon
        return 1
sinon
    si N = ⋀_{i=1}^{k} N_i alors
        return Π_{i=1}^{k} sharpdDnnf(N_i)
    sinon
        //N = ⋁_{i=1}^{k} N_i est alors un nœud ∨ déterministe et uniforme
        return Σ_{i=1}^{k} sharpdDnnf(N_i)
```



La feuille x représente une variable, $\neg x$ sa négation, et α et β sont les racines de formules `NNF`*. On dit que le nœud de décision $N_\vee$ porte sur la variable x et que x est une variable de décision.*

Pour simplifier, on représente souvent graphiquement les nœuds de décision comme sur la partie gauche de la figure 3.7. L'arête en pointillés (respectivement, l'arête pleine) indique que la variable de décision x doit être affectée à faux (respectivement, vrai) dans tout chemin auquel elle participe. Chaque nœud de décision peut être vu comme l'occurrence d'un connecteur ternaire ite (pour *if ... then ... else ...*) dont la sémantique est $ite(x, \alpha, \beta) \equiv (x \wedge \alpha) \vee (\neg x \wedge \beta)$. Pour que $ite(x, \alpha, \beta)$ soit vrai, il faut et il suffit que x soit vrai et alors α doit l'être aussi, et sinon (donc quand $\neg x$ est vrai), β doit être vrai.

Définition 40 (`Decision-DNNF`) *Une formule* `Decision-DNNF` Σ *est une formule* `d-DNNF` *dans laquelle chaque nœud de disjonction est un nœud de décision. Chaque nœud interne d'une formule* `Decision-DNNF` *est donc soit un nœud $\wedge$ décomposable, soit un nœud $\vee$ de décision.*

Clairement, pour chaque chaque nœud de décision d'une formule `Decision-DNNF` Σ où la décision porte sur x, les formules α et β qui lui sont associées ne peuvent pas contenir x. En effet, dans le cas contraire, les nœuds de conjonction de la forme $x \wedge \alpha$ et $\neg x \wedge \beta$ ne seraient pas décomposables, donc Σ ne serait pas une formule `DNNF`.

La figure 3.8 représente une formule `Decision-DNNF`.

Le langage `Decision-DNNF` satisfait les même propriétés que le langage `d-DNNF` (quand on se limite à celles listées plus haut). Les compilateurs existants, tels que *c2d*, ciblant le langage `d-DNNF` ciblent en fait plus spécifiquement le langage `Decision-DNNF`. Par ailleurs, le supplément de structure apportée par les nœud de décision fait qu'il est possible d'évaluer plus finement la taille des formules

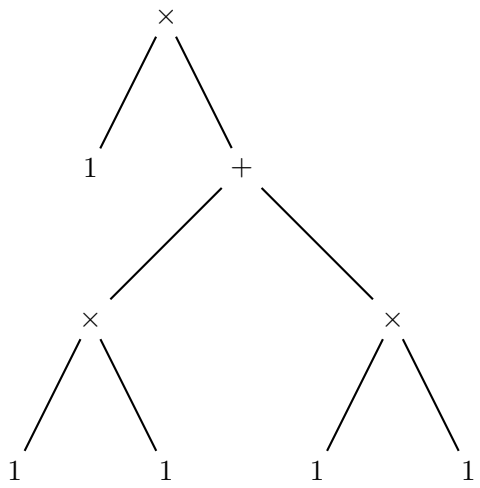


FIGURE 3.6 – Calcul du nombre de modèles d'une formule d−DNNF uniforme

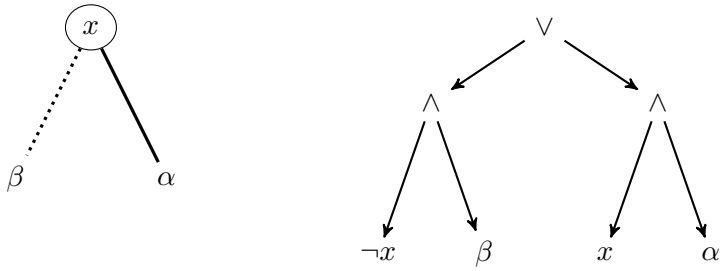


FIGURE 3.7 – Représentation des nœuds de décision

Decision−DNNF et le temps mis pour les calculer, en comparaison aux formules d−DNNF « générales » [OD14].

Une autre restriction du langage d−DNNF est obtenue en important une condition supplémentaire sur la structure des formules. Le langage obtenu s'appelle $\texttt{d−DNNF}_T$ où T est un vtree (c'est-à-dire un arbre sur les variables) [PD08]. Formellement, un vtree sur un ensemble PS de variables propositionnelles est un arbre binaire complet T dont les feuilles sont en bijection avec les variables de PS. Par exemple, la figure 3.9 fournit un vtree sur $PS = \{x, y, z, u, v\}$.

Pour tout nœud interne N de T on note $Var(N^g)$ l'ensemble des variables aux feuilles accessibles à partir du fils gauche de N et $Var(N^d)$ l'ensemble des variables aux feuilles accessibles à partir du fils droit de N.

Une formule d−DNNF Σ dont les nœuds $\wedge$ décomposables sont binaires respecte un vtree T sur $Var(\Sigma)$ si et seulement si pour chaque nœud $\wedge$ décomposable $\alpha \wedge \beta$ de Σ il existe un nœud interne N de T tel que $Var(\alpha) \subseteq Var(N^g)$ et $Var(\beta) \subseteq Var(N^d)$.

Définition 41 ($\texttt{d−DNNF}_T$ **et** sd−DNNF**)** *Le langage* $\texttt{d−DNNF}_T$ *est l'ensemble des formules* d−DNNF *qui respectent un vtree* T *donné. Le langage* sd−DNNF *des* d−DNNF *structurées est l'union des langages* $\texttt{d−DNNF}_T$ *pour tous les vtrees* T *sur* PS.

3.2.2 Les diagrammes de décision binaire

Une autre famille de langages de compilation est celle des diagrammes de décision binaire :

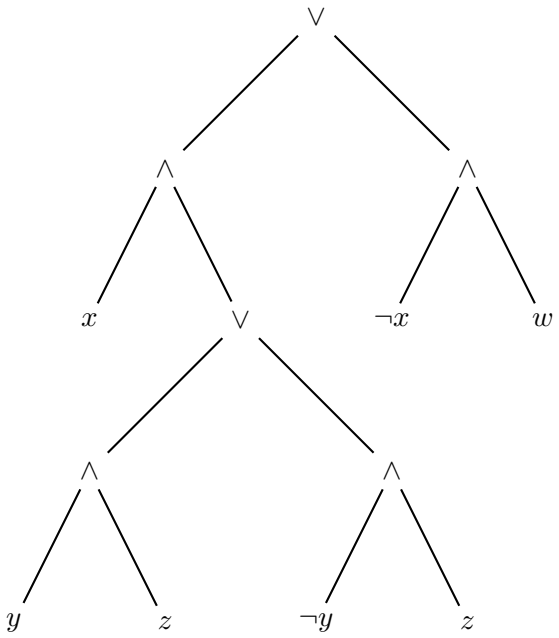


FIGURE 3.8 – Une formule `Decision-DNNF`

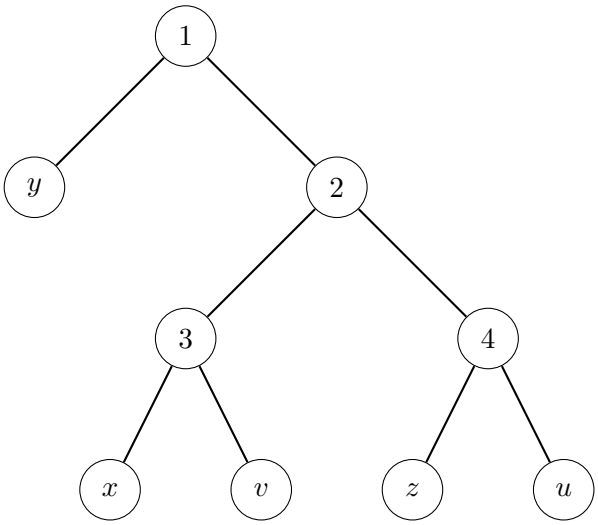


FIGURE 3.9 – Un vtree portant sur l'ensemble des variables $PS = \{x, y, z, u, v\}$

Définition 42 (BDD) *Une formule* BDD Σ *est une formule* NNF *dans laquelle chaque nœud de disjonction est un nœud de décision et qui ne contient pas de nœuds de conjonction autres que ceux imposés par les nœuds de décision.*

Le langage BDD ne peut pas être considéré comme un langage cible pour la compilation, car, comme le langage NNF, il ne vérifie aucune des requêtes prises en compte dans la carte de compilation sauf si P=NP. Cela vient du fait que la décomposabilité des nœuds de conjonction n'est pas systématiquement requise (BDD n'est pas un sous-ensemble de DNNF).

L'ajout de restrictions permet néanmoins d'obtenir des langages intéressants pour la compilation. En particulier, il est intéressant d'exiger que les nœuds de décision soient à lecture unique :

Définition 43 (Lecture unique) *Un nœud de décision* $N_\vee = (x \wedge \alpha) \vee (\neg x \wedge \beta)$ *d'une formule* NNF *est à lecture unique si et seulement si* $x \notin Var(\alpha) \cup Var(\beta)$.

Imposer aux nœuds de décision d'être à lecture unique conduit au langage FBDD :

Définition 44 (FBDD) *Une formule* FBDD Σ *est une formule* BDD *dans laquelle chaque nœud de décision est à lecture unique.*

Cette condition assure que tous les nœuds de conjonction qui figurent dans une formule FBDD Σ sont décomposables. Comme les nœuds de disjonction qui figurent dans Σ sont déterministes ($(x \wedge \alpha) \wedge (\neg x \wedge \beta)$ est contradictoire), il s'ensuit que FBDD est un sous-ensemble de Decision−DNNF. De ce fait, le langage FBDD vérifie toutes les requêtes considérées sauf le test d'implication générale qui n'est pas vérifié sauf si P = NP et le test d'équivalence, dont la satisfaction reste une question ouverte pour le langage FBDD.

Une famille de langages, sous-ensembles de FBDD, a eu un impact important en intelligence artificielle (mais aussi au delà). Il s'agit de la famille des langages des diagrammes de décision binaire ordonnés $\text{OBDD}_<$.

Définition 45 (Ordre) *Soit* $<$ *un ordre total strict sur l'ensemble des variables propositionnelles. Une formule* BDD Σ *respecte* $<$ *si et seulement si pour chaque nœud de décision* $N_\vee = (x \wedge \alpha) \vee (\neg x \wedge \beta)$ *de* Σ *et chaque variable* $y \in Var(\alpha) \cup Var(\beta)$*, on a* $x < y$.

Le langage $\text{OBDD}_<$ induit par $<$ est celui qui contient les formules BDD qui respectent $<$:

Définition 46 ($\text{OBDD}_<$) *Soit* $<$ *un ordre total strict sur l'ensemble des variables propositionnelles. Une formule* $\text{OBDD}_<$ Σ *est une formule* BDD *qui respecte* $<$.

Comme $<$ est strict, chaque langage $\text{OBDD}_<$ est un sous-ensemble de FBDD. L'ajout de la contrainte de compatibilité avec l'ordre $<$ de l'ordre des variables de décision sur chaque chemin de la racine à une feuille permet d'obtenir un langage qui satisfait l'ensemble des requêtes proposées dans la carte de compilation.

L'exemple 40 montre une formule sous forme de diagramme de décision ordonné.

Exemple 40 *Soit* $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$*, avec pour ordre sur les variables* $p < r < q$. Σ *peut être représentée par la formule* $\text{OBDD}_<$ *équivalente, donnée à la figure 3.10.*

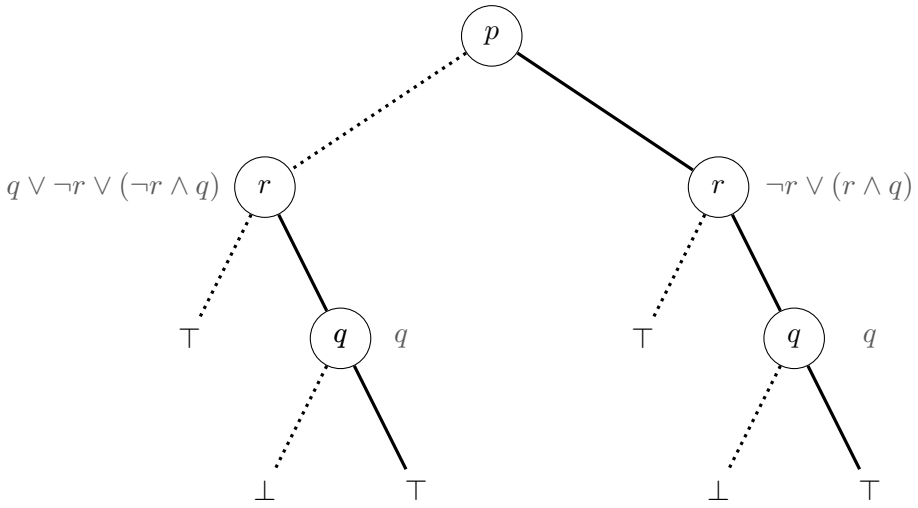


FIGURE 3.10 – Diagramme de décision binaire ordonné représentant la formule Σ.

Une autre propriété des langages $\texttt{OBDD}_<$, qui les rend très attractifs, est l'existence d'algorithmes de réduction qui permettent de réduire la taille de la formule considérée, et surtout d'assurer une propriété de canonicité des représentations réduites : pour tout ordre $<$ sur les variables, deux formules propositionnelles équivalentes possèdent une même représentation sous forme d'une formule $\texttt{OBDD}_<$ quand celle-ci est réduite.

Réduire une formule $\texttt{OBDD}_<$ Σ consiste à appliquer les deux règles de réduction suivantes taat que cela est possible, depuis les feuilles de la formule vers sa racine :

— fusion des nœuds isomorphes : si $(x \wedge N_1) \vee (\neg x \wedge M_1)$ et $(x \wedge N_2) \vee (\neg x \wedge M_2)$ sont deux nœuds de décision distincts de Σ tels que $N_1 = N_2$ et $M_1 = M_2$, alors remplacer dans Σ tout arc ayant le second nœud de décision pour extrémité par un arc ayant le premier nœud de décision pour extrémité ;

— élimination des nœuds redondants : remplacer dans Σ tout nœud de décision de la forme $(x \wedge N) \vee (\neg x \wedge N)$ par N.

L'atout de la canonicité est qu'elle rend faciles certaines requêtes comme le test d'équivalence : pour s'assurer que deux formules propositionnelles représentées par des formules $\texttt{OBDD}_<$ réduites sont équivalentes, il suffit de s'assurer que ces deux dernières formules sont identiques (ce qui peut s'effectuer en temps linéaire dans leur taille via un parcours simultané des deux graphes).

L'exemple 41 donne un exemple de formule $\texttt{OBDD}_<$ réduite.

Exemple 41 *La formule* $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$ *peut être représentée par la formule* $\texttt{OBDD}_<$ *réduite équivalente, donnée à la figure 3.11.*

Par ailleurs, les réductions réalisées permettent parfois de réduire considérablement la taille de la formule compilée.

La famille des langages $\texttt{OBDD}_<$ présente donc un intérêt certain pour la compilation de connaissances. Cependant, la concision des formules obtenues dépend fortement de l'ordre $<$ choisi. Il existe ainsi des (familles de) formules Σ_n (n variant) pour lesquelles la taille de leur représentation en $\texttt{OBDD}_<$ selon un ordre $<_2$ fixé peut être linéaire en la taille de Σ_n, alors que la représentation de Σ_n en $\texttt{OBDD}_<$ pour un autre ordre $<_1$ fixé a une taille exponentielle en la taille de Σ_n.

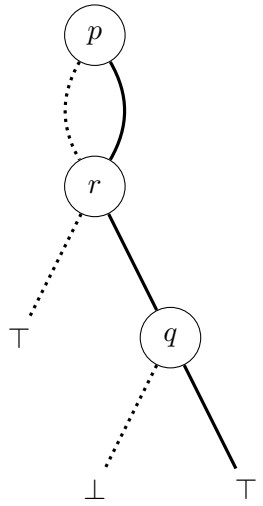


FIGURE 3.11 – Diagramme de décision binaire ordonné réduit de l'exemple 3.10

Exemple 42 *Soit la formule* $\Sigma_n = \bigwedge_{i=1}^{n}(x_i \Leftrightarrow y_i)$*. Si l'ordre défini au départ est* $x_1 <_1 \cdots <_1 x_n <_1 y_1 <_1 \cdots <_1 y_n$*, alors toute représentation en* $\texttt{OBDD}_<$ *de* Σ_n *sera de la forme donnée à la figure 3.12.*

A chacun des n *premiers niveaux de l'ordre* $<_1$*, les nœuds possèdent chacun deux fils et cela, jusqu'au nœud* y_1*. Cette partie de la formule* $\texttt{OBDD}_<$ *ne peut pas être réduite (chaque chemin correspond à une fonction différente des variables* $y_1, \ldots, y_n$*). La prise en compte pour chaque* $i \in \{1, \ldots, n\}$ *de la variable* x_i *conduit à ajouter* 2^{i-1} *nœuds au graphe précédent. De ce fait, la formule obtenue à la fin a une taille en* $\Omega(2^n)$*. A contrario, si l'ordre utilisé est défini comme étant* $x_1 <_2 y_1 <_2 x_2 <_2 y_2 <_2 \cdots <_2 x_n <_2 y_n$*, alors comme pour chaque* $i \in \{1, \ldots, n\}$*,* x_i *et* y_i *doivent être équivalents, il est possible de réduire fortement le graphe résultant : la prise en compte pour chaque* i *de* x_i *puis de* y_i *conduit simplement à ajouter* 4 *nœuds au graphe précédent. De ce fait, la formule obtenue à la fin a une taille en* $O(n)$*. La figure 3.13 décrit (en partie) la formule* $\texttt{OBDD}_<$ *obtenue quand* $<_2$ *est utilisé.*

Le problème de trouver un ordre minimisant la taille de la formule $\texttt{OBDD}_<$ résultante est un problème NP-difficile [SW96].

3.2.3 Les arbres de décision (DT)

Une autre restriction intéressante du langage BDD est obtenue en imposant une contrainte topologique sur les graphes :

Définition 47 (DT) *Un arbre de décision (DT) est une formule BDD arborescente.*

Contrairement au cas du langage BDD, le langage DT est intéressant pour la compilation de connaissances, sans restriction supplémentaire. En particulier, la structure arborescente des graphes considérés Σ permet de ne pas avoir à imposer la condition de lecture seule sur les nœuds de décision. En effet, il suffit de parcourir chaque branche de Σ pour déterminer si elle contient plusieurs occurrences d'un nœud de décision $(x \wedge \alpha) \vee (\neg x \wedge \beta)$ portant sur une même variable x : si tel est le cas, toutes les occurrences de ces nœuds de décision, sauf la première peuvent être éliminées puisqu'elles sont redondantes : si sur le chemin considéré, x est affecté à vrai (resp. faux), il n'y a plus rien à décider ensuite (cette affectation ne peut être remise en cause). Contrairement au cas des BDD, le nombre de branches à parcourir (les chemins de la racine aux feuilles) dans une formule DT Σ est majoré par la taille de cette formule, il est

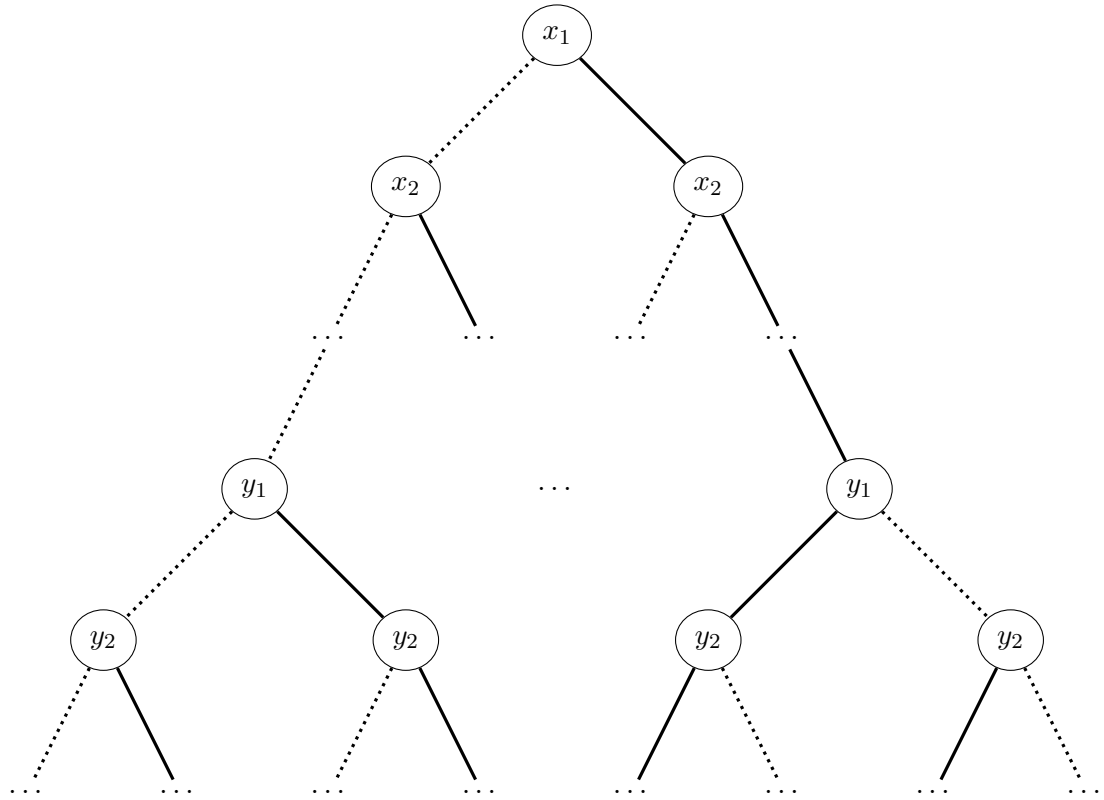


FIGURE 3.12 – Diagramme de décision binaire ordonné de l'exemple 42 suivant l'ordre $x_1 <_1 \cdots <_1 x_n <_1 y_1 <_1 \cdots <_1 y_n$

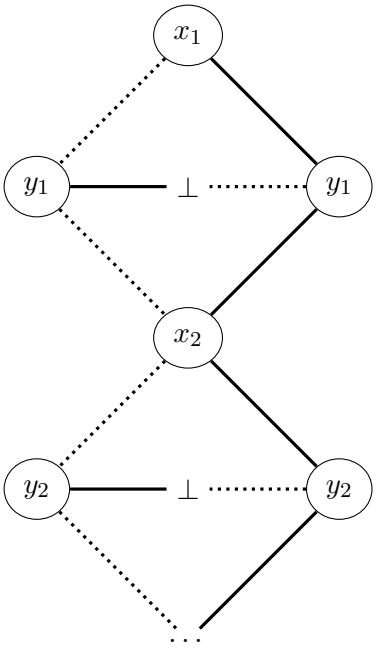


FIGURE 3.13 – Diagramme de décision binaire ordonné de l'exemple 42 suivant l'ordre $x_1 <_2 y_1 <_2 x_2 <_2 y_2 <_2 \cdots <_2 x_n <_2 y_n$

donc possible d'appliquer ce processus de réduction en temps polynomial dans la taille de Σ. Un arbre de décision ainsi réduit est une formule FBDD particulière, dont la forme est arborescente.

Un algorithme simple de construction d'un arbre de décision équivalent à une formule propositionnelle Σ donnée, et qui assure que la condition de lecture seule sur les nœuds de décision est satisfaite, repose sur la décomposition de Shannon :

Définition 48 (Décomposition de Shannon) *Soit une formule propositionnelle Σ. La décomposition de Shannon de Σ sur la variable x est la formule définie par :* $(\neg x \wedge \Sigma_{|x\leftarrow 0}) \vee (x \wedge \Sigma_{|x\leftarrow 1})$.

Comme illustré à la figure 3.14, appliquer la décomposition de Shannon sur une variable x à une formule Σ conduit à créer un nœud de décision portant sur x. Comme, par construction, la variable x n'apparaît pas dans les formules conditionnées $\Sigma_{|x\leftarrow 0}$ et $\Sigma_{|x\leftarrow 1}$, les nœuds de conjonction créés sont décomposables. Enfin, il est facile de montrer que Σ équivaut à sa décomposition de Shannon quelle que soit la variable de décision x choisie.

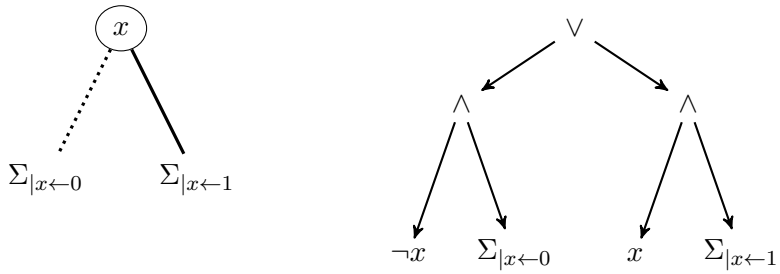


FIGURE 3.14 – Décomposition de Shannon

Pour construire un arbre de décision équivalent à Σ, il suffit d'appliquer la décomposition de Shannon à Σ sur n'importe quelle variable qui apparait dans la formule, puis de réitérer ce processus sur les deux formules $\Sigma_{|x\leftarrow 0}$ et $\Sigma_{|x\leftarrow 1}$ tant que les formules produites par conditionnement contiennent au moins une variable. Comme chaque étape de décomposition conduit à éliminer une variable (x n'apparaît ni dans $\Sigma_{|x\leftarrow 0}$ ni dans $\Sigma_{|x\leftarrow 1}$), le processus va se terminer. Quand $\Sigma_{|x\leftarrow 0}$ ou $\Sigma_{|x\leftarrow 1}$ ne contient plus de variable, il suffit de l'évaluer pour déterminer si la formule équivaut à $\top$ ou si elle équivaut à $\bot$ et d'introduire alors une feuille étiquetée par la constante booléenne ainsi obtenue.

Exemple 43 *Considérons la formule $\Sigma = (q \wedge \neg p) \vee \neg r \vee (((\neg p \wedge \neg r) \vee (p \wedge r)) \wedge q)$. L'arbre représenté à la figure 3.15 est un arbre de décision équivalent à Σ.*

Dans un premier temps, on choisit la variable p. La formule Σ est conditionnée par $\neg p$ à gauche de l'arbre et par p à droite. Une fois la formule conditionnée, elle est simplifiée en suivant les lois du calcul booléen, puis ce processus continue tant que la formule n'est pas équivalente à $\top$ ou $\bot$. Par exemple sur la branche de gauche, cela donne la formule suivante : $\Sigma_{|\neg p} = (q \wedge \top) \vee \neg r \vee (((\top \wedge \neg r) \vee (\bot \wedge r)) \wedge q)$. La formule $(q \wedge \top)$ est équivalente à q, de même $(\top \wedge \neg r)$ est équivalente à $\neg r$, $(\bot \wedge r)$ est équivalente à $\bot$ et donc $((\neg r) \vee (\bot) \wedge q)$ est équivalente à $(\neg r \wedge q)$. Nous obtenons donc comme résultat du conditionnement la formule $\Sigma' = q \vee \neg r \vee (\neg r \wedge q)$. Ensuite, la variable r est sélectionnée. Lorsque l'on considère $\neg r$, on obtient la formule $\Sigma'_{|\neg r} = q \vee \top \vee (\top \wedge q)$ qui est donc équivalente à $\top$, le processus de décomposition s'arrête donc. Puis on considère r, on obtient alors la formule $\Sigma'_{|r} = q \vee \bot \vee (\bot \wedge q)$ qui est logiquement équivalente à la formule $\Sigma'' = q$. Enfin on considère la variable q pour la décomposition de Shannon, Σ'' est donc équivalente à $\bot$ quand $\neg q$ est considéré et à $\top$ quand q est considéré.

Nous remarquons qu'un arbre de décision équivalent à une formule Σ peut être vu comme la juxtaposition d'une représentation de Σ en DNF déterministe avec une représentation de $\neg\Sigma$ en DNF déterministe. En effet, chaque chemin entre la racine d'un arbre de décision et une de ses feuilles représente

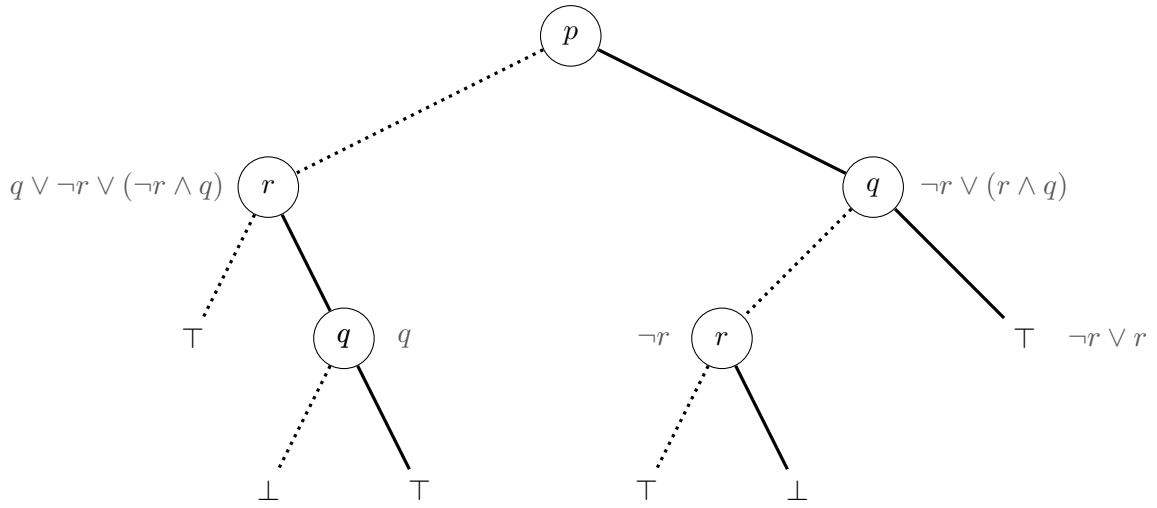


FIGURE 3.15 – Arbre de décision équivalent à la formule Σ de l'exemple 43

un terme (donné par la conjonction des variables de décision rencontrée, chacune munie de la polarité qui convient). Si la feuille atteinte est $\top$, ce terme est un impliquant de la formule Σ, sinon il s'agit d'un impliquant de $\neg\Sigma$.

Exemple 44 *L'arbre de décision de l'exemple précédent peut être vu comme la juxtaposition des deux formules* DNF *déterministes suivantes :*

$$(\neg p \wedge \neg r) \vee (\neg p \wedge r \wedge q) \vee (p \wedge \neg q \wedge \neg r) \vee (p \wedge q)$$

$$(\neg p \wedge r \wedge \neg q) \vee (p \wedge \neg q \wedge r)$$

Il est aisé de montrer que le langage des formules DT Σ dont tous les nœuds de décision sont à lecture seule satisfait **CT**. Il suffit, en effet, de parcourir chaque branche de l'arbre depuis sa racine : lorsque l'on atteint une feuille $\top$, on sait que le terme γ associé est un impliquant de la formule Σ. Donc, à chaque feuille $\top$ de l'arbre de décision, le nombre de modèles de Σ qui est aussi un modèle du terme γ est $2^{|Var(\Sigma)|-|Var(\gamma)|}$. Chaque nœud de décision étant par construction déterministe, deux termes associés à des chemins distincts de Σ ne partagent aucun modèle. II suffit donc d'un parcours de l'arbre Σ pour compter le nombre de ses modèles.

Un problème important posé par le langage DT des arbres de décision est qu'il est très peu performant du point de vue de l'efficacité spatiale. En effet, pour certaines formules propositionnelles à n variables (par exemple $x_1 \oplus \ldots \oplus x_n$, tout arbre de décision équivalent est de profondeur n et possède 2^n feuilles.

3.2.4 Le langage AFF des formules affines

Nous avons considéré jusqu'ici des langages cibles construits à partir des connecteurs $\wedge$, $\vee$, $\neg$ et des constantes booléennes $\top$ et $\bot$. Nous présentons maintenant un langage qui s'appuie sur une autre ensemble de connecteurs logiques, que nous nous retrouverons au chapitre 4.

Ce langage, appelée AFF, inclut des formules dites affines, construites à partir des connecteurs $\wedge$, $\oplus$, $\neg$, $\top$ et $\bot$. On a donc en quelque sorte laissé de côté la disjonction usuelle $\vee$ (inclusive), pour considérer à sa place la disjonction exclusive $\oplus$. Une formule affine est simplement une conjonction finie de clauses

affines. Comme on va le montrer, les formules affines peuvent aussi être considérées comme des systèmes d'équations linéaires sur $\{0, 1\}$, et donc les clauses affines comme des équations linéaires.

Définition 49 (Clause affine (ou XOR-clause)) *Une clause affine, ou XOR-clause, est une formule de la forme $l_1 \oplus \cdots \oplus l_n$ où $l_1, \ldots, l_n$ sont des littéraux ou des constantes booléennes. La taille de cette formule est n.*

Définition 50 (Formule affine `AFF`) *Une formule affine est une conjonction finie de clauses affines (de XOR-clauses).*

Exemple 45 *La formule suivante est une formule affine :*

$$(x_2 \oplus \neg x_4 \oplus \neg x_5 \oplus \top) \wedge (x_1 \oplus \neg x_2 \oplus x_3) \wedge (x_1 \oplus \neg x_4 \oplus \top)$$

Dans le but de simplifier l'écriture et de ne pas surcharger les formules, il est possible de normaliser l'écriture de chaque clause affine (on le fera assez systématiquement dans la suite) :

Définition 51 (Forme normale des clauses affines) *Une clause affine simplifiée, ou clause affine sous forme normale, est une clause affine constituée de littéraux positifs portant sur des variables distinctes et d'au plus une occurrence de $\top$.*

Toute clause affine peut être transformée en une clause affine simplifiée en temps linéaire, en utilisant les équivalences $\neg x \equiv x \oplus \top$, $x \oplus x \equiv \bot$, $\top \oplus \top \equiv \bot$, $x \oplus \bot \equiv x$ et en profitant du fait que $\oplus$ est un connecteur associatif et commutatif.

Ainsi, la formule affine donnée à l'exemple 45 peut être transformée en temps linéaire en la formule affine suivante, dite simplifiée, i.e., dans laquelle toutes les clauses affines ont été simplifiées :

Exemple 46 *La formule suivante est une formule affine simplifiée qui équivaut à la formule affine donnée à l'exemple 45 :*

$$(x_2 \oplus x_4 \oplus x_5 \oplus \top) \wedge (x_1 \oplus x_2 \oplus x_3 \oplus \top) \wedge (x_1 \oplus x_4)$$

Chaque clause affine simplifiée de la forme $x_1 \oplus x_2 \oplus \cdots \oplus x_n$ (respectivement, $x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus \top$) peut aussi être vue comme une équation linéaire de la forme $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ (respectivement, $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 0$). Il suffit en effet de voir $=$ comme le symbole d'équivalence $\Leftrightarrow$, 1 comme la constante $\top$ et 0 comme la constante $\bot$ et de se souvenir que $\alpha \Leftrightarrow \beta$ équivaut à $\alpha \oplus \beta \oplus \top$: les affectations de valeur (de vérité) aux variables qui satisfont l'équation $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 1$ (respectivement, $x_1 \oplus x_2 \oplus \cdots \oplus x_n = 0$) sont exactement celles qui satisfont la clause associée.

Notons qu'une clause affine peut être représentée de façon canonique sous forme `CNF` par la conjonction de toutes les clauses formées sur l'ensemble de ses variables, contenant un nombre pair (respectivement un nombre impair) de littéraux négatifs si la clause affine contient un nombre pair (respectivement un nombre impair) de littéraux négatifs.

Exemple 47 *La formule affine $(\neg x_2 \oplus x_4 \oplus x_5) \wedge (x_1 \oplus x_2 \oplus x_3) \wedge (\neg x_1 \oplus x_4)$ est équivalente à la formule `CNF` suivante :*

$$\begin{array}{llll} \neg x_2 & \vee & x_4 & \vee & x_5 \\ x_2 & \vee & \neg x_4 & \vee & x_5 \\ x_2 & \vee & x_4 & \vee & \neg x_5 \\ \neg x_2 & \vee & \neg x_4 & \vee & \neg x_5 \end{array} \qquad \begin{array}{llll} x_1 & \vee & x_2 & \vee & x_3 \\ \neg x_1 & \vee & \neg x_2 & \vee & x_3 \\ \neg x_1 & \vee & x_2 & \vee & \neg x_3 \\ x_1 & \vee & \neg x_2 & \vee & \neg x_3 \end{array} \qquad \begin{array}{lll} \neg x_1 & \vee & x_4 \\ x_1 & \vee & \neg x_4 \end{array}$$

Clairement, nous remarquons que la formule affine initiale est beaucoup plus concise que sous sa forme CNF. En effet, la représentation d'une clause affine sous forme CNF est exponentielle en la taille de la clause affine. Plus précisément, une clause affine simplifiée de taille n ne possède que des représentations CNF qui contiennent au moins 2^{n-1} clauses.

Un atout important du langage AFF est qu'il satisfait **CT** [Zan03]. En fait, l'ensemble des requêtes considérées dans la carte de compilation peuvent être résolues efficacement sur les formules affines Tout est basé sur un même algorithme : l'algorithme d'élimination de Gauss/Jordan, aussi appelé pivot de Gauss.

Le pivot de Gauss est un algorithme permettant de déterminer l'ensemble des solutions d'un système d'équations linéaires. Il consiste ici à réduire le nombre de clauses affines de la formule affine considérée Σ pour faire en sorte d'obtenir une formule affine logiquement équivalente, possédant un nombre minimum de clauses affines. L'algorithme du pivot de Gauss permet ainsi de garantir que la formule affine résultante Σ_t contient au maximum $\mid Var(\Sigma) \mid$ clauses affines. L'algorithme 5 représente l'algorithme du pivot de Gauss. L'algorithme prend en entrée une formule affine Σ et donne en sortie une formule affine Σ_t. Le principe de l'algorithme est de trianguler la formule affine Σ selon un ordre total et strict $<$ sur les variables. Par exemple, la formule AFF $(x \oplus y \oplus u \oplus w) \wedge (y \oplus z \oplus w \oplus \top) \wedge (z \oplus u)$ est triangulée puisque la première clause affine contient x, la clause affine suivante contient y mais pas x, et la dernière clause affine contient z mais pas les variables x et y. Nous pouvons ainsi représenter la formule sous forme d'une matrice triangulaire de la forme :

$$\begin{pmatrix} x & y & & u & w & \\ & y & z & & w & \top \\ & & z & u & & \end{pmatrix}$$

Intuitivement, l'idée de l'algorithme est de faire en sorte d'ordonner les clauses de sorte que une variable (la plus petite selon $<$) de la première clause affine n'apparaisse pas dans les clauses affines suivantes. Puis, que la seconde clause contienne une variable (la seconde plus petite selon $<$) qui n'apparaisse pas dans les prochaines clauses affines, et ainsi de suite pour toutes les variables de la formule affine. À la fin de l'exécution nous obtenons donc une formule affine qui possède au plus n clauses affines, avec n le nombre de variables de la formule en entrée.

Plus formellement, le pivot de Gauss pivotGauss décrit à l'algorithme 5 procède comme suit. Dans un premier temps, ligne 1 la formule résultat Σ_t est initialisée à l'ensemble vide de clauses. Ensuite, tant que la formule Σ n'est pas vide, une variable p de Σ est sélectionnée, ligne 3 ainsi qu'une clause affine c_a de Σ contenant p, ligne 4. Cette variable sera la variable *pivot* pour la clause c_a choisie, c'est-à-dire la variable qui sera éliminée des clauses restantes dans Σ. Ligne 5 et 6, la clause affine sélectionnée est ajoutée à la formule de sortie et supprimée de Σ. Ensuite, ligne 7, pour toute clause affine c'_a de Σ contenant la variable pivot p, c'_a est remplacée par le résultat de l'addition modulo 2 de cette clause avec c_a, noté $c'_a \oplus c_a$. Comme p apparaît dans les deux clauses, l'addition modulo 2 de ces deux clauses affines a pour effet de supprimer p dans la clause résultat.

Exemple 48 *soit $c_{a1} = x \oplus y \oplus u$ et $c_{a2} = y \oplus w \oplus u \oplus \top$ deux clauses affines, l'addition modulo 2 des deux clauses donne $c_{a1} \oplus c_{a2} = x \oplus w$. Nous remarquons que lorsque les clauses affines c_{a1} et c_{a2} contiennent une même variable, celle-ci disparaît de $c_{a1} \oplus c_{a2}$.*

Lorsque les clauses ont la même parité de négations (respectivement une parité différente), alors le résultat de l'addition modulo 2 de ces deux clauses possède un nombre impair (respectivement pair) de négations.

Enfin, la formule alors triangulée est retournée ligne 9. Nous notons que le résultat de l'addition modulo 2 entre deux clauses affines simplifiées dépend du nombre de négations (donc de la présence de la constante $\top$) dans ces clauses.

Algorithme 5 : pivotGauss$(\Sigma, <)$

entrée : une formule AFF Σ et un ordre $<$ total strict sur $Var(\Sigma)$
sortie : une formule AFF Σ_t triangulée selon l'ordre $<$ équivalente à Σ

```
Σ_t ← ∅;
while Σ ≠ ∅ do
    p ← la plus petite variable de Σ selon <;
    c_a ← une clause de Σ contenant p;
    Σ_t ← Σ_t ∪ {c_a};
    Σ ← Σ \ {c_a};
    forall c'_a ∈ Σ tel que p ∈ c'_a do
        Σ ← Σ \ {c'_a};
        Σ ← Σ ∪ {c'_a ⊕ c_a};
return Σ_t;
```

Exemple 49 *Soit la formule affine* $\Sigma = (x \oplus y \oplus z) \wedge (x \oplus u \oplus w) \wedge (y \oplus z \oplus w)$. *Appliquons l'algorithme du pivot de Gauss sur cette formule, avec comme ordre sur les variables* $x < y < z < u < w$. *Représentons la formule affine* Σ *sous forme de matrice :*

$$\begin{pmatrix} x & y & z & & \\ x & & & u & w \\ & y & z & & w \end{pmatrix} \quad \begin{matrix} (c_{a1} \in \Sigma) \\ (c_{a2} \in \Sigma) \\ (c_{a3} \in \Sigma) \end{matrix}$$

En premier lieu, la variable x *est choisie comme pivot p, et nous choisissons la clause affine* c_{a1} *pour la variable* c_a *de l'algorithme 5. Nous supprimons* c_{a1} *de* Σ *et nous l'ajoutons dans* Σ_t. *La variable* x *apparaît dans la clause affine* c_{a2}, *nous remplaçons donc* c_{a2} *par le résultat de l'addition modulo 2 entre* c_{a2} *et* c_{a1}. *Nous obtenons la matrice suivante :*

$$\begin{pmatrix} x & y & z & & & & \top \\ & y & z & u & w & & \\ & y & z & & w & & \end{pmatrix} \quad \begin{matrix} (c_{a1} \in \Sigma_t) \\ (c'_{a2} = c_{a2} \oplus c_{a1} \in \Sigma) \\ (c_{a3} \in \Sigma) \end{matrix}$$

La variable y *est ensuite sélectionnée comme nouvelle variable pivot et la clause* c'_{a2} *est choisie pour la variable* c_a *de l'algorithme 5. Nous supprimons* c'_{a2} *de* Σ *et nous l'ajoutons dans* Σ_t. *Comme* y *apparaît dans* c_{a3} *nous remplaçons donc* c_{a3} *par le résultat de l'addition modulo 2* $c'_{a2} \oplus c_{a3}$.

$$\begin{pmatrix} x & y & z & & & & \top \\ & y & z & u & w & & \\ & & & u & & & \end{pmatrix} \quad \begin{matrix} (c_{a1} \in \Sigma_t) \\ (c'_{a2} \in \Sigma_t) \\ (c_{a3} \oplus c'_{a2} \in \Sigma) \end{matrix}$$

Enfin, il ne reste plus qu'une clause affine, l'algorithme va la sélectionner et l'ajouter à Σ_t. *La formule* Σ *est alors vide, donc l'algorithme retourne la formule affine* $\Sigma_t = (x \oplus y \oplus z) \wedge (y \oplus z \oplus u \oplus w \oplus \top) \wedge (u)$ *qui est triangulée.*

Dans l'exemple, les variables x, y et u sont les variables choisies successivement comme variables pivots. Dans la formule affine triangulée Σ_t, nous disons que ces variables sont les variables liées et les autres variables z et w sont les variables libres. Lorsque la formule est triangulée nous remarquons que les affectations possibles des variables liées x, y et u dans ses modèles dépendent uniquement des valeurs des variables libres z et w dans ceux-ci.

Dans cet exemple, nous avons utilisé un ordre particulier $<$ d'élimination pour les variables. Cependant, l'ordre retenu n'influe pas sur la correction de l'algorithme. Celui-ci reste correct peu importe l'ordre choisi, et l'ordre retenu ne change pas sa complexité. Seule la formule résultat pourra être différente selon l'ordre choisi, mais toutes les formules résultats possibles sont logiquement équivalentes à la formule donnée en entrée et ont le même nombre de clauses.

L'algorithme du pivot de Gauss est en temps polynomial en la taille de la formule en entrée [Zan03]. Plus précisément, cet algorithme calcule une formule affine triangulée Σ_t logiquement équivalente à Σ en temps $\mathcal{O}(\mid \Sigma \mid^2 \, . \mid Var(\Sigma) \mid)$. De plus, la formule affine Σ_t possède au plus $\mid Var(\Sigma) \mid$ clauses affines.

Nous notons que le plus petit système d'équation linéaire ne possédant aucune solution est $0 = 1$, qui correspond à la formule $\bot$. Si la formule affine Σ en entrée ne possède aucun modèle, l'algorithme du pivot de Gauss retourne donc en sortie la formule $\bot$. L'algorithme du pivot de Gauss permet donc de décider de la cohérence d'une formule affine Σ en temps $\mathcal{O}(\mid \Sigma \mid^2 \mid Var(\Sigma) \mid)$.
Si la formule Σ est cohérente, nous pouvons facilement calculer un modèle de celle-ci à partir de la formule triangulée Σ_t. En effet, comme nous l'avons remarqué précédemment, dans chaque clause de la formule affine triangulée obtenue, il y a une variable liée, dont la valeur de vérité ne dépend que des autres variables de la formule. L'idée est donc de procéder de façon gloutonne à partir de la dernière clause affine de la formule. Nous affectons arbitrairement les variables libres de cette clause, puis nous remontons à la clause suivante, et ainsi de suite. Lorsque toutes les variables libres sont assignées, les variables liées sont assignées par propagation.

Exemple 50 *Soit la formule affine triangulée Σ_t de l'exemple précédent. Nous souhaitons calculer un modèle deΣ_t. Pour cela, nous affectons la variable u de la dernière clause à $\top$ puisqu'ici u est unitaire. Puis, cette valeur est propagée dans la deuxième clause que nous réduisons pour obtenir alors la clause $y \oplus z \oplus w$. Nous pouvons choisir d'affecter $\top$ aux variables w et z et de ce fait y doit prendre la valeur $\top$. Enfin, ces valeurs sont remontées à la première clause que l'on simplifie. Cette clause devient alors la clause unitaire x. La variable x doit donc prendre pour valeur $\top$. Nous obtenons donc l'interprétation complète $\{x, y, z, u, w\}$ qui est bien un modèle de la formule Σ de l'exemple considéré.*

Plus généralement, pour connaître le nombre de modèles d'une formule affine triangulée, il suffit de connaître le nombre de ses variables libres. Celui-ci est facilement calculable. En effet, il y a exactement une variable liée pour chaque clause de la formule affine triangulée et celle-ci contient un nombre minimal de clauses. Par conséquent, le nombre de variables libres de Σ_t vaut le nombre de variables de Σ_t moins le nombre de clauses affines de celle-ci.

Définition 52 (Nombre de modèles d'une formule affine) *Soit Σ une formule affine et Σ_t une formule affine triangulée logiquement équivalente à Σ. Si Σ_t contient k variables libres, alors Σ possède 2^k modèles (sur $Var(\Sigma)$).*

Nous disposons donc d'un algorithme en temps polynomial qui permet de calculer le nombre de modèles d'une formule affine donnée. D'autres requêtes liées au comptage de modèles (le test de cohérence, celui de validité) sont évidemment offertes en prime, à titre de conséquence. Ceci rend le langage `AFF` très compétitif vu nos objectifs.

Cependant, le langage `AFF` des formules affines n'est pas complet pour la logique propositionnelle. En effet, certaines formules propositionnelles ne peuvent pas être représentées par des conjonctions de formules affines. Par exemple, la formule $x_1 \vee x_2$ ne peut pas être représentée par une conjonction de clauses affines. Ce langage n'est donc pas utilisable dans de nombreux cas, à cause de son expressivité réduite.

Pour pallier ce problème, il faut généraliser le langage AFF. Nous pouvons alors par exemple le généraliser pour autoriser la présence de certaines disjonctions (inclusives) dans les formules. Ainsi, un langage constitué de disjonctions de formules affines AFF[$\vee$] a-t-il été proposé dans [FM08b]. Il a été montré qu'un tel langage est complet pour la logique propositionnelle. Cependant, l'ajout de la disjonction à ce langage implique la perte de quelques transformations telles $\wedge$**C** et $\neg$**C** ainsi que la perte de beaucoup de requêtes, telles **VA**, **IM**, **EQ**, **SE** et le comptage de modèles **CT** qui est la principale requête qui nous intéresse.

Nous présenterons au chapitre prochain d'autres généralisations de AFF permettant de pallier son incomplétude, tout en préservant **CT**.

3.3 Résultats de la carte de compilation

Le tableau 3.1 résume les requêtes satisfaites par les langages que nous avons présentés. Nous remarquons que le langage NNF ne satisfait aucune requête considérée sauf si P=NP. Le langage CNF ne vérifie que le test de validité et le test d'impliquant. Ces deux langages ne sont donc pas intéressants en tant que langages cibles pour la compilation. Le langage AFF satisfait toutes les requêtes, cependant il n'est pas complet pour la logique propositionnelle et ne peut donc pas être considéré comme un langage cible pour la compilation exacte de certaines formules. Les langages d−DNNF et OBDD$_<$ sont deux langages satisfaisant le comptage de modèles (**CT**) et donc les plus intéressants (parmi ceux présentés plus haut) quand on se focalise sur cette requête.

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
NNF	○	○	○	○	○	○	○	○
CNF	○	√	○	√	○	○	○	○
DNF	√	○	√	○	○	○	○	√
DNNF	√	○	√	○	○	○	○	√
d−DNNF	√	√	√	√	?	○	√	√
FBDD	√	√	√	√	?	○	√	√
OBDD$_<$	√	√	√	√	√	√	√	√
AFF	√	√	√	√	√	√	√	√
AFF[$\vee$]	√	○	√	○	○	○	○	√

TABLE 3.1 – Requêtes. √ signifie "satisfait" et ○ signifie "ne satisfait pas sauf si P = NP."

Le tableau 3.2 résume les transformations que vérifient les langages que nous avons présentés. Nous remarquons que tous ces langages satisfont le test de conditionnement (**CD**). Le langage NNF vérifie toutes les transformations sauf l'oubli (**FO**). Le langage d−DNNF ne vérifie que le conditionnement, tandis que le langage OBDD$_<$ vérifie toutes les transformations sauf l'oubli, la conjonction et la disjonction.

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
NNF	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
CNF	$\surd$	$\circ$	$\surd$	$\surd$	$\surd$	$\bullet$	$\surd$	$\bullet$
DNF	$\surd$	$\surd$	$\surd$	$\bullet$	$\surd$	$\surd$	$\surd$	$\bullet$
DNNF	$\surd$	$\surd$	$\surd$	$\circ$	$\circ$	$\surd$	$\surd$	$\circ$
d-DNNF	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?
FBDD	$\surd$	$\bullet$	$\circ$	$\bullet$	$\circ$	$\bullet$	$\circ$	$\surd$
$\text{OBDD}_<$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
AFF	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	!	!	!
AFF[$\vee$]	$\surd$	$\circ$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\surd$

TABLE 3.2 – Transformations. $\surd$ signifie “satisfait”, $\bullet$ signifie “ne satisfait pas” et $\circ$ signifie “ne satisfait pas sauf si P = NP”.

Chapitre 4

Les arbres de décision affine

Sommaire

4.1 Les réseaux de décision affine (ADN) 68
4.1.1 Les graphes de décision affine (ADG) et les arbres de décision affine étendus généralisés 71
4.1.2 Le langage des arbres de décision affine étendus ($\text{EADT}_\wedge$) et ses sous-ensembles 73
4.2 Extension de la carte de compilation 77
4.2.1 Les requêtes 78
4.2.2 Les transformations 81
4.2.3 Efficacité spatiale 89
4.3 Un compilateur CNF-vers-$\text{EADT}_\wedge$ 91
4.3.1 Décompositions de Shannon 91
4.3.2 Algorithme de compilation 93
4.3.3 Expérimentations 97
4.4 Conclusion 98

Dans ce chapitre, nous définissons une nouvelle famille de langages cibles pour la compilation de connaissances, offrant le comptage de modèles. Nous nous focalisons sur trois langages particuliers : le langage des arbres de décision affine étendus ($\text{EADT}_\wedge$), et ses sous-ensembles, à savoir le langage des arbres de décision affine (ADT) et le langage des arbres de décision étendus (EDT). Comme leurs noms l'évoquent, ces langages ont en commun le fait de regrouper des formules dont la forme est arborescente. Chacun de ces langages est un sur-ensemble du langage DT défini au chapitre précédent.

Dans la suite de ce chapitre, nous définissons formellement chacun de ces langages. Nous étudions leurs propriétés de façon à pouvoir les situer dans la carte de compilation. Nous comparons ces nouveaux langages aux langages principaux déjà placés dans la carte de compilation et offrant **CT**. Nous décrivons ensuite un algorithme permettant de compiler une formule CNF en $\text{EADT}_\wedge$. L'efficacité de cet algorithme a été évaluée expérimentalement et nous présentons les résultats expérimentaux obtenus. Sur cette base, nous avons pu comparer la compilation en $\text{EADT}_\wedge$ à d'autres approches de compilation ainsi qu'un compteur de modèles direct faisant état de l'art. Ce travail a donné lieu à une publication dans les actes de la conférence IJCAI 2013 [KLMT13].

Dans la carte de compilation des formules propositionnelles, peu de langages cibles offrent la requête **CT**, c'est-à-dire sont munis d'algorithmes en temps polynomial pour compter les modèles d'une formule. Parmi eux figurent principalement les langages d-DNNF et ses sous-ensembles, Decision-DNNF, $\text{OBDD}_<$, mais aussi DT, ainsi que le langage représentant l'ensemble des modèles MODS. Comme nous l'avons déjà évoqué, MODS n'est que rarement utilisable en pratique, il est en effet un langage peu efficace d'un point de vue de l'efficacité spatiale. Un langage intéressant pour le comptage de modèles est DT,

le langage des arbres de décision ; cependant, il est lui aussi peu efficace dun point de vue spatial. Le langage AFF des formules affines permet le comptage de modèles en temps polynomial, mais il n'est pas complet pour la logique propositionnelle : il souffre donc d'un déficit d'expressivité, ce qui rend son utilisation délicate (que faire si l'information disponible n'est pas représentable comme une formule de AFF ?).

L'idée directrice qui a conduit à la définition du langage ADT des arbres de décision affine a été de combiner les principes qui sont à la base des formules affines (l'utilisation de clauses affines) et des arbres de décision (la prise en compte de nœuds de décision organisés en arbre). Pour généraliser les arbres de décision, nous étendons les nœuds de décision standard à des nœuds de décision affine, c'es-à-dire associés à des clauses affines. Le langage $\texttt{EADT}_\wedge$ des arbres de décision affine étendus a ensuite été défini en autorisant dans les formules la présence de nœuds de conjonction vérifiant une forme forte de décomposabilité. Enfin, le langage EDT des arbres de décision étendus peut être vu comme la restriction de $\texttt{EADT}_\wedge$ au cas où les nœuds de décision sont tous standard.

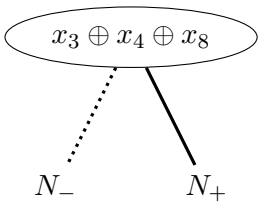
4.1 Les réseaux de décision affine (ADN)

Commençons par définir un langage très général, le langage ADN des réseaux de décision affine, qui va servir de langage de base pour définir la sémantique des formules de ces sous-ensembles qui vont plus particulièrement nous intéresser, ADT, $\texttt{EADT}_\wedge$, et EDT. Tout comme NNF, ADN n'est pas un langage cible intéressant car il ne vérifie pas **CT**. Son introduction sert essentiellement à fixer un cadre et des notions utiles.

ADN (et ses sous-ensembles ADT et $\texttt{EADT}_\wedge$) se base sur un nouveau type de nœud, les nœuds de décision affine, définis comme suit :

Définition 53 (Nœud de décision affine) *Une nœud de décision affine est un nœud de décision binaire de la forme $N = \langle \delta, N_-, N_+ \rangle$ où δ est la clause affine étiquetant N et N_- (respectivement N_+) est le fils gauche (respectivement droit) de N. N_- représente la branche dans laquelle la clause affine $\delta \oplus \top$ sera considérée comme vraie, et N_+ la clause affine δ sera considérée comme vraie.*

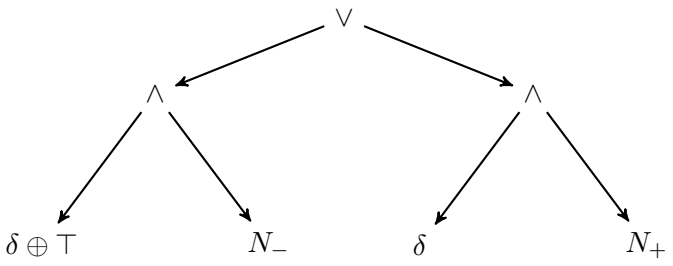
Exemple 51 *Soit $N = \langle \delta, N_-, N_+ \rangle$ le nœud de décision affine représenté par la figure suivante :*



La clause affine $x_3 \oplus x_4 \oplus x_8$ est la clause affine δ de N. Le fils gauche du nœud, représenté en pointillés, N_- est le fils où la clause affine $\delta \oplus \top$ sera considérée comme vraie. Dans notre cas, la clause affine considérée comme vraie dans N_- est donc $x_3 \oplus x_4 \oplus x_8 \oplus \top$. Les formules affines Σ formées sur le fils gauche de ce nœud contiendront cette clause affine, autrement dit, nous avons $\Sigma \wedge (x_3 \oplus x_4 \oplus x_8 \oplus \top)$. Le fils droit, représenté par un trait plein, N_+ est le fils où la clause affine δ est considérée comme vraie. Dans notre cas, la clause $x_3 \oplus x_4 \oplus x_8$ sera prise en compte dans N_+.

Comme pour les nœuds de décision habituels, un nœud de décision affine correspond à une disjonction déterministe de deux conjonctions :

Exemple 52 *Le nœud de décision affine $N = \langle \delta, N_-, N_+ \rangle$ correspond à la formule propositionnelle suivante :*



Un réseau de décision affine est un graphe dont les nœuds internes sont des nœuds de disjonction, de conjonction ou de décision affine. Le langage contenant ces réseaux est défini formellement de la manière suivante :

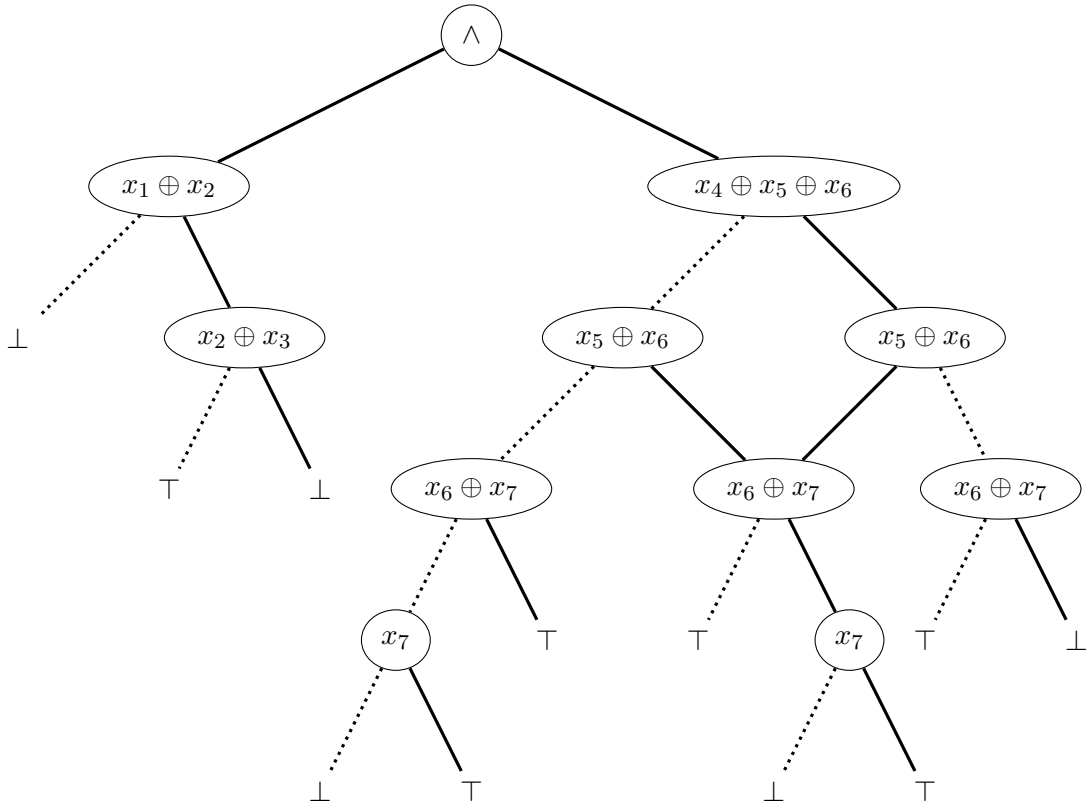
Définition 54 (ADN**)** *Un réseau de décision affine (*ADN*) est un graphe orienté fini sans circuit (DAG) ne comportant qu'une racine, où les feuilles sont étiquetées par $\top$ ou $\bot$, et les nœuds internes sont des nœuds de conjonction ou de disjonction (avec un nombre arbitraire mais fini de fils) ou des nœuds de décision affine.*

Afin de comparer les différents langages en terme d'efficacité spatiale, il est nécessaire de définir la notion de taille d'une formule. Alors que la taille d'une formule NNF est donnée par son nombre d'arcs, la taille $|\Delta|$ d'une formule ADN Δ est définie de la manière suivante :

Définition 55 (Taille d'un ADN**)** *La taille $|\Delta|$ d'une formule* ADN Δ *est la somme de son nombre d'arcs et de la taille cumulée des clauses affines utilisées pour étiqueter les nœuds de décision affine.*

Nous notons R_Δ la racine du graphe représentant une formule ADN Δ. Soit Δ une formule ADN et N un nœud de Δ, nous notons Δ_N la sous-formule de Δ représentée par la formule ADN ayant pour nœud racine N.

Exemple 53 *Soit la formule propositionnelle Σ suivante : $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_5 \vee x_7)$. La formule* ADN Δ *suivante est une représentation de Σ sous forme* ADN*.*



Nous remarquons que la formule Σ définie dans l'exemple 53 est décomposable. En effet, nous pouvons la décomposer en deux formules Σ_1 et Σ_2 portant sur deux sous-ensembles disjoints de $Var(\Sigma)$. Nous avons $Var(\Sigma_1) = \{x_1, x_2, x_3\}$ et $Var(\Sigma_2) = \{x_4, x_5, x_6, x_7\}$. Les deux formules Σ_1 et Σ_2 peuvent donc être transformées respectivement en deux formules ADN logiquement équivalentes Δ_1 et Δ_2 puis être reliées par un nœud $\wedge$ décomposable.

La racine R_Δ de la formule ADN Δ de l'exemple 53 est un nœud $\wedge$ décomposable. Cette formule possède 22 arcs et 19 littéraux. Donc $|\Delta| = 22 + 19 = 41$.

Pour tout nœud N d'une formule ADN Δ, l'ensemble $Var(N)$ des variables constituant la formule ADN Δ_N enracinée en N, est défini de manière inductive comme :

— si N est une feuille, alors $Var(N) = \varnothing$;
— si N est un nœud de décision affine $N = \langle \delta, N_-, N_+ \rangle$, alors $Var(N) = var(\delta) \cup Var(N_-) \cup Var(N_+)$;
— si N est un nœud $\wedge$ (respectivement un nœud $\vee$) ayant comme fils $N_1, \ldots, N_k$, alors $Var(N) = \cup_{i=1}^{k} Var(N_i)$.

L'ensemble $Var(\Delta)$ des variables d'une formule ADN Δ est ainsi égal à l'ensemble des variables de la racine R_Δ de Δ. $Var(R_\Delta)$ peut être calculé en temps linéaire en la taille de Δ. En effet, ce calcul peut être effectué par un simple parcours du graphe Δ.

Exemple 54 *Soit la formule* ADN Σ *de l'exemple précédent, l'ensemble des variables de la formule* ADN $\Sigma_{x_1 \oplus x_2}$*, la sous-formule de* Σ *dont le nœud racine est le nœud* $x_1 \oplus x_2$*, est* $Var(\Sigma_{x_1 \oplus x_2}) = \{x_1, x_2, x_3\}$*. De même, nous avons* $Var(\Sigma) = Var(R_\Sigma) = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$.

Toute formule ADN Δ est interprétée comme une formule propositionnelle $I(\Delta)$ sur les variables $Var(\Delta)$, avec $I(\Delta) = I(R_\Delta)$ défini de manière inductive par :

— si N est une feuille étiquetée par $\top$ (respectivement $\bot$), alors $I(N) = \top$ (respectivement $\bot$) ;

— si N est un nœud de décision affine $N = \langle \delta, N_-, N_+ \rangle$, $I(N) = ((\delta \oplus \top) \wedge I(N_-)) \vee (\delta \wedge I(N_+))$;
— si N est un nœud $\wedge$ (respectivement $\vee$) ayant pour fils $N_1, \ldots, N_k$, alors $I(N) = \bigwedge_{i=1}^{k} I(N_i)$ (respectivement $\bigvee_{i=1}^{k} I(N_i)$).

Notons qu'un nœud de décision affine étiqueté par la clause affine $c_a = x_{i1} \oplus \cdots \oplus x_{in}$ divise les interprétations possibles d'une formule en deux parties égales. En effet, nous aurons d'un côté un fils dans lequel les seules interprétations possibles seront celles dont un nombre pair de variables de $\{x_{i1}, \ldots, x_{in}\}$ seront assignées à vrai, et celles dont un nombre impair de variables de $\{x_{i1}, \ldots, x_{in}\}$ seront assignées à vrai.

Exemple 55 *Soit la formule* ADN Σ *de l'exemple 53. Les interprétations complètes*

$$\mathcal{I}_1 = \{x_1, \neg x_2, x_3, \neg x_4, x_5, x_6, x_7\}$$
$$\mathcal{I}_2 = \{x_1, x_2, x_3, x_4, \neg x_5, x_6, \neg x_7\}$$

sont respectivement modèle et contre-modèle de Σ.

Nous rappelons que le nombre de modèles d'une formule ADN Δ sur $Var(\Delta)$ est noté $\|\Delta\|$. Par exemple, le nombre de modèles de la formule considérée dans l'exemple 53 précédent est $\|\Sigma\| = 22$.

Il est important de noter que le langage DAG−NNF est traduisible en temps polynomial en un sous-ensemble de ADN, dans lesquels les nœuds de décision affine ne possèdent que des feuilles comme fils. En effet, chaque nœud étiqueté par un littéral positif x (respectivement négatif $\neg x$) d'une formule DAG−NNF est équivalent à un nœud de décision affine N étiqueté par x, tel que $N_- = \bot$ (respectivement $N_- = \top$) et $N_+ = \top$ (respectivement $N_+ = \bot$).

En conséquence, le langage ADN est un langage de représentation très succinct. A contrario, il offre peu de requêtes parmi celles considérées dans la carte de compilation. En effet, comme il est possible de transformer en temps polynomial n'importe quelle formule DAG−NNF en formule ADN, si une requête n'est pas offerte par DAG−NNF, elle ne l'est pas non plus pour ADN. Or, nous avons vu au chapitre 3.2 que DAG−NNF n'offre aucune requête de la carte de compilation.

Ainsi, ADN ne satisfait pas le comptage de modèles **CT** (sauf si P = NP) ni même le test de cohérence **CO**.

C'est pourquoi nous allons par la suite nous concentrer sur des sous-ensembles du langage ADN.

4.1.1 Les graphes de décision affine (ADG) et les arbres de décision affine étendus généralisés

Restreindre les nœuds de conjonction et de disjonction (qui ne participent pas à des nœuds de décision) à être des nœuds décomposables conduit au langage ADG des graphes de décision affine :

Définition 56 (ADG) *Une formule* ADG *est une formule* ADN *dans laquelle les nœuds internes sont des nœuds de décomposition affine et des nœuds de conjonction décomposables.*

Imposer en plus aux formules considérées d'avoir une forme arborescente conduit au langage GEADT des arbres de décision affine étendus généralisés :

Définition 57 (GEADT) *Une formule* GEADT *est une formule* ADG *dont la forme est arborescente.*

La restriction qui consiste à ne considérer dans les formules que des nœuds de décision et des nœuds de conjonction décomposables conduit, quand on l'applique au langage NNF, au langage Decision-DNNF, qui est un langage cible intéressant. Elle ne suffit toutefois pas à définir un langage cible qui a de l'intérêt quand on les applique au langage ADN. En effet, ni le langage ADG, ni son sous-ensemble GEADT ne satisfait le test de cohérence :

Proposition 3 GEADT *ne satisfait pas* **CO**.

Preuve 1 *Soit* $\Sigma = \bigwedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ *une formule 3-*CNF *quelconque. Nous montrons qu'il est possible de traduire* Σ *en temps polynomial en une formule* GEADT *qui est équivalente à* Σ*. Soit* α *la formule affine donnée par* $\alpha = \bigwedge_{v \in Var(\Sigma)} \bigwedge_{i=1}^{n}(v \oplus \neg v^i)$*. Autrement dit, pour toute variable* v *de* Σ*, nous créons* n *nouvelles variables* v^i *logiquement équivalentes à* v*.*

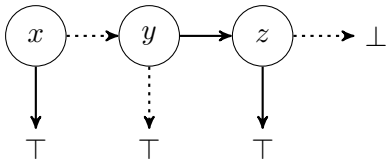
Soit Δ *la formule* CNF *telle que* $\Delta = \bigwedge_{i=1}^{n}(l^i_{i,1} \vee l^i_{i,2} \vee l^i_{i,3})$*.* Δ *peut être vue comme la formule 3-*CNF Σ *dans laquelle chaque occurrence de variable est renommée pour être indicée par le rang de la clause où elle apparaît.* Δ *est donc une conjonction décomposable de clauses. Par ailleurs, il est possible de transformer toute clause en une formule* DT *(donc* GEADT*) en temps polynomial (voir l'exemple 56). En effet, pour cela, il suffit de considérer un littéral* $l = x$ *(respectivement* $l = \neg x$*) de la clause, de créer un nœud de décision* $\langle x, N_-, N_+ \rangle$*, où* N_+ *(respectivement* N_-*) est la feuille* $\top$ *et* N_- *est la racine d'une représentation dans* DT *de la clause privée du littéral* l *(la clause vide étant représentée par la feuille* $\bot$*). Ainsi,* Δ *peut être traduite en temps polynomial en une formule équivalente qui s'écrit comme une conjonction décomposable de formules* DT*.*

Comme α *est un formule affine, nous pouvons facilement la traduire, elle aussi, en temps polynomial en une formule* GEADT *qui ne contient pas de nœud de conjonction (hormis ceux correspondant aux nœud de décision affine). Il suffit de considérer la formule obtenue en choisissant d'abord une clause affine* $\alpha_i = v \oplus v^i$ *de* α*, puis en créant un un nœud de décision affine* $\langle v \oplus v^i, N_-, N_+ \rangle$*, où* $N_- = \bot$ *et* N_+ *est la racine d'une formule de même nature, représentant la formule* α *privée de la clause affine* α_i *(et pour la dernière clause affine de* α *considérée, on définit* $N_+ = \top$*). Voir encore l'exemple 56.*

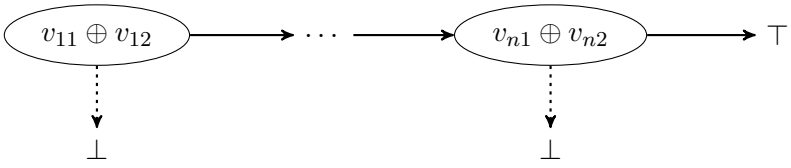
Remarquons maintenant que Σ *est cohérente si et seulement si* $\alpha \wedge \Delta$ *est cohérente.*

Enfin, la conjonction des représentations en GEADT *de ces deux formules peut être réalisée en temps polynomial : il suffit de remplacer chaque feuille* $\top$ *de la représentation de* α *par la racine d'une copie de la représentation en* GEADT *de* Δ*. Nous avons donc défini un algorithme en temps polynomial pour traduire toute formule 3-*CNF Σ *en une formule* GEADT *qui est cohérente si et seulement si* Σ *est cohérente. Cela prouve que le langage* GEADT *ne satisfait pas le test de cohérence* **CO** *sauf si* P = NP*.*

Exemple 56 *Soit* $x \vee \neg y \vee z$ *une clause de taille* 3*. Une formule* DT *équivalente est la suivante :*



Soit $\alpha = \bigwedge_{i=1}^{n}(v_{i1} \oplus v_{i2})$ *une formule affine composée de* n *clauses affines. La formule suivante est une représentation* GEADT *de* α *qui ne contient pas de nœud de conjonction :*



A fortiori, le langage ADG ne satisfait pas la requête **CO**. Par conséquent, ni GEADT ni ADG ne satisfont **CT**.

4.1.2 Le langage des arbres de décision affine étendus ($\text{EADT}_\wedge$) et ses sous-ensembles

Pour obtenir un langage cible intéressant, il est donc nécessaire d'imposer à ADN des restrictions supplémentaires à celles considérées pour définir le langage GEADT. L'une d'entre elles consiste à renforcer la condition de décomposabilité portant sur les nœuds de conjonction et elle conduit au langage $\text{EADT}_\wedge$ des arbres de décision affine étendus.

Les arbres de décision affine étendus ($\text{EADT}_\wedge$)

Le langage $\text{EADT}_\wedge$ s'appuie sur la notion de nœud de conjonction affine décomposable, défini comme suit :

Définition 58 (Nœud affine décomposable) *Soit un nœud de conjonction N ayant pour fils $N_1, \ldots, N_k$ dans une formule ADN Δ. N est affine décomposable si et seulement si :*

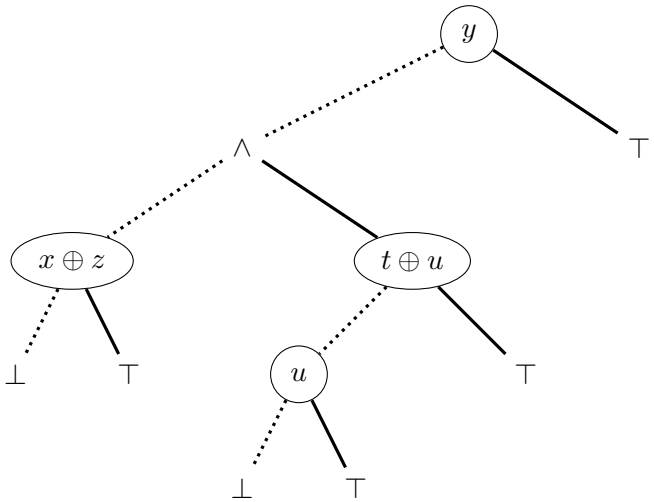
(1) pour tout $i, j \in 1, \ldots, k$, si $i \neq j$, alors $Var(N_i) \cap Var(N_j) = \varnothing$, et

(2) il existe au plus un fils N_i de N tel que pour tout nœud de décision affine $N' = \langle \delta, N_-, N_+ \rangle$ de Δ qui est un ancêtre de N et pour lequel la clause affine δ n'est pas réduite à un littéral, nous avons $Var(N_i) \cap var(\delta) \neq \varnothing$.

Si nous ne prenons en compte que la première condition de la définition des nœuds affines décomposables, alors le nœud N est (classiquement) décomposable.

Définition 59 ($\text{EADT}_\wedge$) *Une formule $\text{EADT}_\wedge$ est une formule GEADT dont les nœuds de conjonction (qui ne participent pas aux nœuds de décision) sont affines décomposables.* [2]

Exemple 57 *Par exemple, la formule CNF $\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$ est logiquement équivalente à la formule $\text{EADT}_\wedge$ suivante :*



2. Ce langage est la restriction du langage EADT défini dans [KLMT13] pour laquelle les nœuds $\vee$ décomposables ne sont pas utilisés (Nous faisons le choix de nous focaliser sur cette restriction dans la mesure où notre compilateur cible précisément le langage $\text{EADT}_\wedge$).

Nous montrons un peu plus loin dans le chapitre que $\texttt{EADT}_\wedge$ vérifie **CT**. Avant cela, introduisons deux restrictions utiles du langage $\texttt{EADT}_\wedge$.

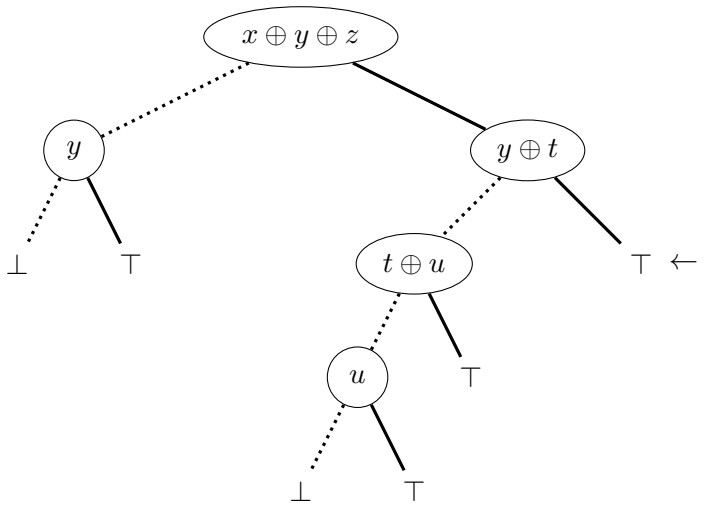
Les arbres de décision affine (ADT)

Les formules de ADT sont les arbres de décision affine :

Définition 60 (ADT) *Une formule* ADT *est une formule* $\texttt{EADT}_\wedge$ *dans laquelle on ne trouve aucun nœud de conjonction, sauf ceux issus des nœuds de décision.*

Ainsi, il est possible de considérer ce langage comme une extension du langage DT dans laquelle les nœuds de décision standard sont généralisés en des nœuds de décision affine.

Exemple 58 *La formule* CNF $\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$ *est logiquement équivalente à la formule* ADT *suivante :*



Chaque chemin entre la racine R_Δ d'une formule ADT Δ et une feuille de Δ est associé à une formule affine, la conjonction des clauses affines rencontrées sur le chemin. À titre d'illustration, sur l'exemple 58, considérons le chemin conduisant à la feuille repérée par une flèche : comme la feuille est étiquetée par $\top$, la formule affine qui correspond à ce chemin est $(x \oplus y \oplus z) \wedge (y \oplus t)$. Tout nœud de décision affine étant, par définition, un nœud de disjonction déterministe, une formule ADT peut être vue comme une disjonction déterministe de formules affines.

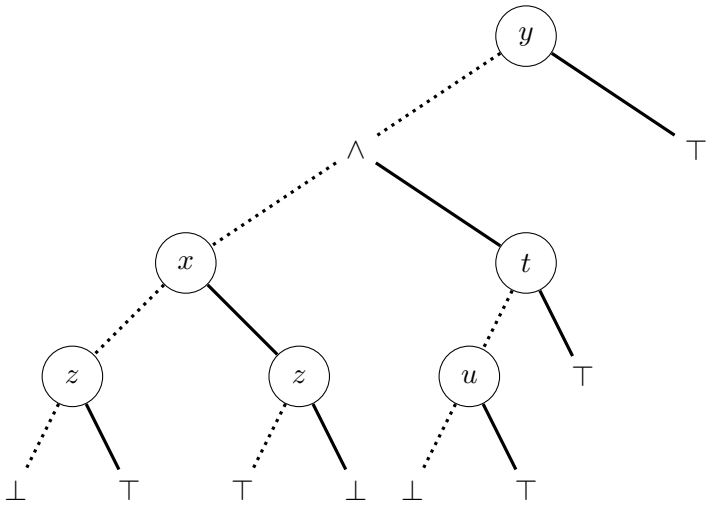
Les arbres de décision étendus (EDT)

Le langage EDT des arbres de décision étendu est un autre sous-ensemble de $\texttt{EADT}_\wedge$, dans lequel les nœuds internes sont soit des nœuds de décision affine unitaire (i.e., des nœuds de décision classiques) soit des nœuds de conjonction classiquement décomposables :

Définition 61 (EDT) *Une formule* EDT *est une formule* $\texttt{EADT}_\wedge$ *dans laquelle tous les nœuds de décision sont des nœuds de décision standard.*

Le langage EDT peut donc être vu comme une extension du langage DT dans laquelle les nœuds internes des formules peuvent aussi être des nœuds de conjonction décomposables.

Exemple 59 *La formule* CNF $\Sigma = (\neg x \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (y \vee t \vee u)$ *est logiquement équivalente à la formule* EDT *suivante :*



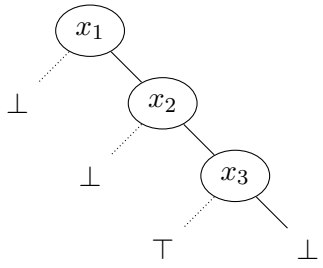
Le langage EDT peut donc aussi être vu comme la restriction du langage Decision-DNNF (que nous avons présenté au chapitre 3), aux formules dont la forme est arborescente.

Quelques propriétés du langage $\text{EADT}_\wedge$ et de ses sous-ensembles

Nous mettons maintenant en évidence quelques propriétés simples du langage $\text{EADT}_\wedge$ et de ses sous-ensembles, qui concernent leur efficacité spatiale et leur expressivité. Tout d'abord, nous prouvons que pour le langage des termes TE et celui des clauses CL (tous deux définis au chapitre 3.2), il existe un algorithme de traduction en temps polynomial vers le langage DT, donc a fortiori vers ses sur-ensembles ADT, EDT et $\text{EADT}_\wedge$.

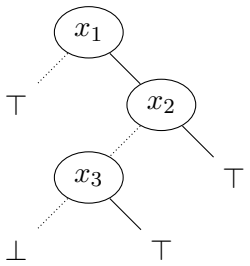
Considérons d'abord le langage TE des termes. Soit γ un terme quelconque. Si γ est le terme vide ou la constante $\top$, γ équivaut à la formule DT réduite à la feuille $\top$. Si γ contient $\bot$, γ équivaut à la formule DT réduite à la feuille $\bot$. Sinon nous choisissons un littéral l de variable x apparaissant dans γ, nous créons un nœud de décision $N = \langle x_i, N_-, N_+ \rangle$. Si la polarité de x dans γ est positive (respectivement, négative), alors N_- (respectivement, N_+) est la feuille $\bot$ et N_+ (respectivement, N_-) est la racine d'une représentation DT du terme obtenu en retirant l de γ. Cet algorithme de construction est clairement en temps linéaire dans la taille de γ.

Exemple 60 *Par exemple, voici le terme* $x_1 \wedge x_2 \wedge \neg x_3$ *une fois compilé en formule* DT *:*

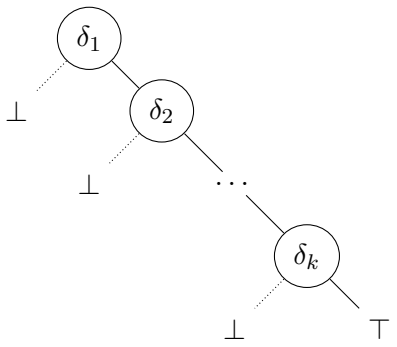


Par dualité, l'algorithme de traduction de CL vers DT est similaire. Soit δ une clause quelconque. Si δ est la clause vide ou la constante $\bot$, δ équivaut à la formule DT réduite à la feuille $\bot$. Si δ contient $\top$, δ équivaut à la formule DT réduite à la feuille $\top$. Sinon nous choisissons un littéral l de variable x apparaissant dans δ, nous créons un nœud de décision $N = \langle x_i, N_-, N_+ \rangle$. Si la polarité de x dans δ est positive (respectivement, négative), alors N_+ (respectivement, N_-) est la feuille $\top$ et N_- (respectivement, N_+) est la racine d'une représentation DT de la clause obtenue en retirant l de δ. Cet algorithme de construction est clairement en temps linéaire dans la taille de δ.

Exemple 61 *La clause* $\neg x_1 \vee x_2 \vee x_3$ *correspond à la formule* DT *suivante :*



Le langage AFF est lui polynomialement traduisible vers ADT, donc vers $\text{EADT}_\wedge$. En effet, comme nous l'avons remarqué précédemment, tout chemin entre la racine R_Δ d'un ADT Δ et une feuille de Δ représente une formule affine. Donc une formule affine $\Sigma = \bigwedge_{i=1}^{k} \delta_i$ où chaque δ_i ($i\ in 1, \ldots, k$) est une clause affine, est équivalente à la formule ADT sous forme de « peigne » suivante :



Pour construire une telle formule ADT à partir de Σ, Il suffit en effet de parcourir la formule affine et pour chaque clause affine δ_i, nous créons un nœud $N = \langle \delta_i, N_-, N_+ \rangle$ où N_- est la feuille $\bot$ et N_+ est la racine d'une représentation ADT de la formule affine obtenue en retirant δ_i de Σ (et la conjonction vide de clauses affines est représentée par la feuille $\top$ comme il se doit).

Contrairement aux langages TE, CL et AFF, le langage DT et ses sur-ensembles ADT, EDT et $\text{EADT}_\wedge$ sont des langages complets pour la logique propositionnelle. Cela vient simplement du fait que le langage des arbres de décision binaire « standard » DT est un langage complet et qu'il est un sous-ensemble de ADT, de EDT, et donc de $\text{EADT}_\wedge$.

La propriété de complétude est aussi satisfaite par le langage des arbres de décision ordonnés ($\text{ODT}_<$), le sous-ensemble de DT des formules Δ où chaque chemin de la racine aux feuilles de Δ respecte un ordre strict et total $<$ sur l'ensemble des variables. Il apparaît clairement que $\text{ODT}_<$ est également un sous-ensemble de $\text{OBDD}_<$ (plus précisément, $\text{ODT}_<$ est l'intersection des langages DT et $\text{OBDD}_<$). Il est aisé de montrer que les langages CL et TE sont polynomialement traduisibles vers $\text{OBDD}_<$. Comme $\text{ODT}_<$ est un sous-ensemble de $\text{OBDD}_<$, le langage des arbres de décision réduits possède une propriété intéressante de canonicité : deux formules logiquement équivalentes ont la même représentation en $\text{ODT}_<$, une

fois réduite (où seule la règle d'élimination des variables redondantes est appliquée). Cette propriété de canonicité n'est pas offerte par les langages DT, ADT, EDT et EADT$_\wedge$.

4.2 Extension de la carte de compilation

Dans cette section, nous analysons les langages introduits dans la section précédente, i.e., le langage des arbres de décision affine (ADT), celui des arbres de décision étendus (EDT) et celui des arbres de décision affine étendus (EADT$_\wedge$), selon les critères utilisés dans la carte de compilation des formules propositionnelles. Nous nous efforçons donc de déterminer si ces langages satisfont ou non les requêtes et des transformations proposées dans la carte de compilation [DM02]. Nous évaluons aussi l'efficacité spatiale de ces langages.

Les langages des arbres de décision (DT) et des arbres de décision ordonnés (ODT$_<$) quoique bien connus n'ont pas été insérés dans la carte de compilation. Pour cette raison et vu la proximité avec les langages introduits, nous avons décidé de les ajouter à notre étude.

Montrons tout d'abord comment toute formule EADT$_\wedge$ Δ peut être traduite en temps linéaire en un arbre $T(\Delta)$ où les nœuds internes sont des nœuds $\wedge$ décomposables ou des nœuds $\vee$ déterministes à deux fils, et les feuilles sont étiquetées par des formules affines. La première étape consiste à propager de la racine vers les feuilles les valeurs de vérité des variables de décision, i.e., celles qui apparaissent dans des nœuds de décision standard. Plus précisément, si $N = \langle x, N_-, N_+\rangle$ est un nœud de décision de Δ alors toute occurrence de x dans la sous-formule de Δ enracinée en N_- (respectivement, N_+) est remplacée par $\bot$ (respectivement, $\top$) (et les clauses affines ainsi modifiées sont simplifiées). Une fois cela réalisé, on est sûrs qu'aucune occurrence de x n'existe dans toute formule enracinée à un nœud qui est un descendant d'un nœud de décision portant sur x.

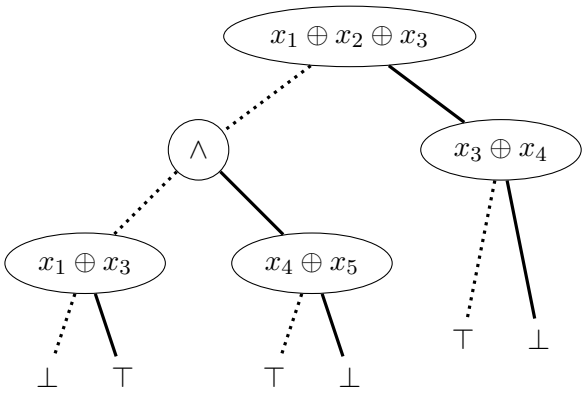
La seconde étape consiste à propager de la racine vers les feuilles de l'arbre obtenu les clauses affines que l'on rencontre. Formellement, on associe à chaque nœud de l'arbre un attribut hérité a dont la valeur est l'ensemble des clauses affines collectées. Pour la racine de l'arbre, cet attribut vaut $\varnothing$. Si $N = \langle \delta, N_-, N_+\rangle$ est un nœud de décision affine, $\delta \oplus \top$ est propagée vers N_- et δ vers N_+, c'est-à-dire $a(N_-) = a(N) \cup \{\delta \oplus \top\}$ et $a(N_+) = a(N) \cup \{\delta\}$. N est alors remplacé par un nœud de disjonction (qui par construction va être déterministe). Si N est un nœud de conjonction $\bigwedge_{i=1}^{k} N_i$ qui est donc forcément affine décomposable, alors s'il existe un fils N_j de N tel que $Var(N_j) \cap a(N) \neq \varnothing$, on propage les clauses collectées vers N_j, c'est-à-dire $a(N_j) = a(N)$ et pour tout autre fils N_i de N ($i \in \{1, \ldots, k\}, i \neq j$), on a $a(N_i) = \varnothing$; sinon, on propage les clauses collectées vers N_1, c'est-à-dire $a(N_1) = a(N)$ et pour tout autre fils N_i de N ($i \in \{1, \ldots, k\}, i \neq 1$), on a $a(N_i) = \varnothing$. Le nœud de conjonction N est conservé.

Une fois cela effectué, l'arbre $T(\Delta)$ est obtenu en remplaçant chaque feuille N étiquetée par $\top$ par la formule affine qui est la conjonction des clauses de $a(N)$ et en conservant chaque feuille N étiquetée par $\bot$ (qui est une formule affine).

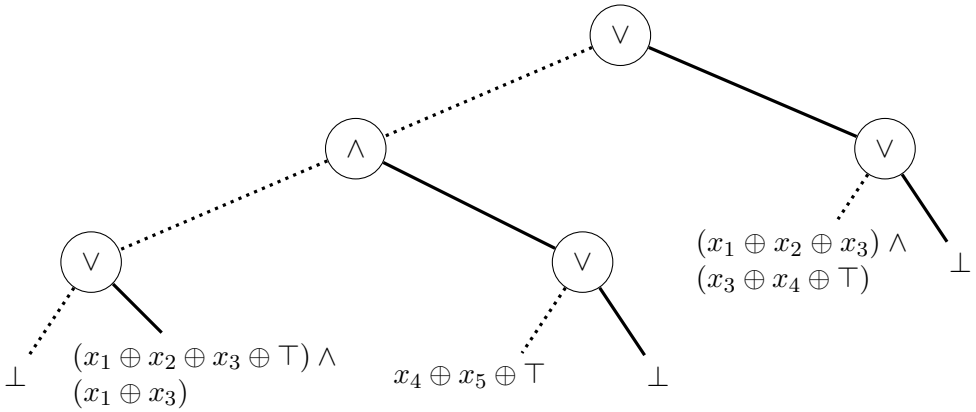
Les langages ADT et EDT étant tous deux des sous-ensembles de EADT$_\wedge$, la traduction T peut aussi être appliquée à leurs éléments. Quand Δ est une formule ADT, $T(\Delta)$ est simplement une disjonction déterministe de formules affines et quand Δ est une formule EDT, $T(\Delta)$ est simplement une formule DNF déterministe.

L'exemple suivant illustre la façon dont la traduction T opère.

Exemple 62 *Soit la formule* $\texttt{EADT}_\wedge\ \Delta$ *suivante :*



La formule $T(\Delta)$ *obtenue est représentée par l'arbre suivant :*



4.2.1 Les requêtes

Nous commençons par déterminer les requêtes qu'offrent les langages considérés. Comme $\texttt{EADT}_\wedge$ (respectivement ADT) est un sur-ensemble de EDT, ADT, DT et $\texttt{ODT}_<$ (respectivement, de DT et $\texttt{ODT}_<$), chaque fois que $\texttt{EADT}_\wedge$ satisfait une requête, cette requête est aussi satisfaite par tous ses sous-ensembles. On utilisera aussi la contraposée de cette implication : lorsqu'un sous-ensemble d'un langage donné n'offre pas une requête, ce langage ne satisfait pas non plus la requête.

Le comptage de modèles (CT)

Dans un premier temps, nous montrons que le langage des arbres de décision affine étendus permet le comptage de modèles. En s'appuyant sur la transformation T précédente, il est facile de montrer que $\texttt{EADT}_\wedge$ satisfait la requête **CT**.

Proposition 4 $\texttt{EADT}_\wedge$ *satisfait* **CT**.

Preuve 2 *La preuve est par induction structurelle sur la formule* $T(\Delta)$ *obtenue après traduction en temps polynomial de la formule* $\texttt{EADT}_\wedge\ \Delta$. *Premièrement, nous avons vu au chapitre 3 que le nombre de modèles* $\|\phi\|$ *d'une formule affine* ϕ *peut être calculé en temps polynomial. Cela résout le cas de base où* ϕ *est une formule affine. Ensuite, par induction :*

— *si* $\phi = \bigwedge_{i=1}^{k} \phi_i$ *(où* $\bigwedge$ *est un nœud* $\wedge$ *affine décomposable), alors*

$$\|\phi\| = \prod_{i=1}^{k} \|\phi_i\|$$

— *si* $\phi = \phi_1 \vee \phi_2$ *(où* $\vee$ *est un nœud* $\vee$ *déterministe), alors*

$$\|\phi\| = \|\phi_1\| \times 2^{|Var(\phi_2) \setminus Var(\phi_1)|} + \|\phi_2\| \times 2^{|Var(\phi_1) \setminus Var(\phi_2)|}$$

Le calcul de $\|T(\Delta)\| = \|\Delta\|$ s'effectue donc en ordre topologique inverse, des feuilles de $T(\Delta)$ vers sa racine. Chaque feuille de l'arbre est associée à une formule affine. Il suffit donc d'appliquer l'algorithme du pivot de Gauss à afin d'en déterminer le nombre de modèles. Ensuite, nous remontons ce nombre aux ancêtres des nœuds jusqu'à la racine de l'arbre (formellement, on calcule la valeur d'un attribut synthétisé). Si le nœud père est un nœud $\wedge$ décomposable, alors nous multiplions le nombre retourné par chacun de ses fils et ce nombre est retourné à son père. Sinon c'est un nœud $\vee$ déterministe : nous normalisons le nombre de modèles retourné par ses fils et nous additionnons les nombres de modèles et ce nombre est retourné à son père. Ce processus est itéré jusqu'à la racine de $T(\Delta)$ et le nombre obtenu à la racine représente bien le nombre de modèles de la formule $\texttt{EADT}_{\wedge}\ \Delta$.

Nous avons donc un algorithme en temps polynomial permettant de calculer le nombre exact de modèles d'une formule $\texttt{EADT}_{\wedge}$. En conséquence, le langage $\texttt{EADT}_{\wedge}$ et ses sous-ensembles, ADT, EDT, DT et $\texttt{ODT}_{<}$ offrent **CT**, le comptage du nombre de modèles.

La cohérence (CO) et la validité (VA)

Une conséquence triviale du fait que $\texttt{EADT}_{\wedge}$ et ses sous-ensembles, ADT, EDT, DT et $\texttt{ODT}_{<}$ satisfont **CT** est qu'ils satisfont aussi **CO** et **VA**. En effet, la requête de cohérence consiste à déterminer si le nombre de modèles est différent de zéro. La requête de validité, quant à elle, revient à déterminer si le nombre de modèles d'une formule Σ portant sur n variables est égal à 2^n.

L'implication clausale (CE)

Un langage vérifie le test d'implication clausale si et seulement si il existe un algorithme en temps polynomial qui, pour toute formule Σ du langage et pour toute clause δ, retourne vrai si $\Sigma \models \delta$ et faux sinon. Si δ est une clause valide (ce que l'on peut tester en temps polynomial), alors $\Sigma \models \delta$ est vérifié. On peut donc s'intéresser seulement au cas où δ n'est pas valide. On sait que l'on a $\Sigma \models \delta$ si et seulement si $\Sigma \wedge \neg\delta$ est contradictoire. La requête de l'implication clausale, $\Sigma \models \delta$, revient donc à déterminer si la formule $\Sigma \wedge \neg\delta$ est incohérente. Comme δ est une clause de la forme $l_1 \vee \cdots \vee l_k$, la formule $\neg\delta$ équivaut à un terme γ de la forme $\bar{l}_1 \wedge \cdots \wedge \bar{l}_k$, où $\bar{l}_i$ ($i \in \{1, \ldots, k\}$) est le littéral complémentaire de l_i. On a donc $\Sigma \models \delta$ si et seulement si $\Sigma \wedge \gamma$ est cohérente. De plus, γ est un terme cohérent puisque δ n'est pas valide. On sait aussi que $\Sigma \wedge \gamma$ équivaut à $(\Sigma \mid \gamma) \wedge \gamma$. Comme cette conjonction est décomposable et que γ est cohérent, on a $(\Sigma \mid \gamma) \wedge \gamma$ cohérent si et seulement si $\Sigma \mid \gamma$ est cohérent. De ce fait, tout langage vérifiant le test de cohérence **CO** et le conditionnement **CD** satisfait l'implication clausale [DM02]. Puisque le langage $\texttt{EADT}_{\wedge}$ satisfait le test de cohérence et, comme nous le verrons ensuite, satisfait aussi **CD** (voir 4.2.2), $\texttt{EADT}_{\wedge}$ offre la requête de l'implication clausale **CE**. Cette requête est donc aussi offerte par ses sous-ensembles ADT, EDT, DT et $\texttt{ODT}_{<}$.

Le test d'impliquant (IM)

Un langage vérifie le test d'impliquant si et seulement si il existe un algorithme en temps polynomial qui, pour toute formule Σ du langage et pour tout terme γ, retourne vrai si $\gamma \models \Sigma$ et faux sinon. Si le

terme γ est incohérent, alors on sait que $\gamma \models \Sigma$ est toujours $vrai$. Nous nous intéressons donc au cas où γ est cohérent. Vérifier si un terme cohérent γ implique une formule Σ, revient à vérifier si la formule $\neg\gamma \vee \Sigma$ est valide. Or, la formule $\neg\gamma \vee \Sigma$ est logiquement équivalente à la formule $\neg\gamma \vee (\gamma \wedge \Sigma)$. Comme nous l'avons rappelé ci-dessus, γ étant un terme, la formule $\gamma \wedge \Sigma$ est équivalente à $\gamma \wedge (\Sigma \mid \gamma)$. La formule $\neg\gamma \vee \Sigma$ est donc équivalente à $\neg\gamma \vee (\gamma \wedge (\Sigma \mid \gamma))$, qui équivaut encore à $(\neg\gamma \vee \gamma) \wedge (\neg\gamma \vee (\Sigma \mid \gamma))$, donc à $\neg\gamma \vee (\Sigma \mid \gamma)$. Comme $\Sigma \mid \gamma$ et $\neg\gamma$ ne partagent aucune variable, leur disjonction est valide si et seulement si $\Sigma \mid \gamma$ est valide ou $\neg\gamma$ est valide. Comme γ est cohérent, $\neg\gamma$ ne peut pas être valide. Par conséquent, déterminer si la formule $\neg\gamma \vee \Sigma$ est valide revient à tester si $\Sigma \mid \gamma$ est valide.

Donc si le langage considéré permet le conditionnement en temps polynomial et satisfait le test de validité en temps polynomial, nous pouvons tester en temps polynomial la validité de $\Sigma \mid \gamma$. Autrement dit, tout langage vérifiant le test de validité **VA** et le conditionnement **CD** satisfait aussi le test d'implication clausale [DM02]. Comme le langage $\texttt{EADT}_\wedge$ satisfait le test de validité et offre le conditionnement, il offre le test d'impliquant **IM**. En conséquence, les langages ADT, EDT, DT et $\texttt{ODT}_<$ satisfont aussi **IM**.

L'implication générale (SE)

Le test d'implication générale **SE** est satisfait par un langage si et seulement si pour toute formule Σ_1 et Σ_2 de ce langage, il existe un algorithme en temps polynomial qui retourne vrai si $\Sigma_1 \models \Sigma_2$, faux sinon. Nous ne savons pas si $\texttt{EADT}_\wedge$ et EDT satisfont **SE**. Nous démontrons en revanche que le langage des ADT et ses sous-langages DT et $\texttt{ODT}_<$ satisfont le test d'implication général en temps polynomial.

Ce résultat vient directement du fait qu'il offre les transformations de conjonction bornée et de négation et qu'il offre aussi le test de cohérence [DM02]. En effet, soient Σ_1 et Σ_2 deux formules ADT. Nous savons que déterminer si Σ_2 est impliquée par Σ_1 est équivalent à déterminer si la formule $\Sigma_1 \wedge \neg\Sigma_2$ est incohérente. Or, le langage ADT satisfait les transformations de négation et de conjonction bornée et offre le test de cohérence. Par conséquent, le langage ADT et ses sous-ensembles DT et $\texttt{ODT}_<$ satisfont le test d'implication générale.

L'équivalence (EQ)

Le problème de déterminer si deux formules des langages $\texttt{EADT}_\wedge$ ou EDT sont logiquement équivalentes reste lui aussi un problème ouvert. Nous ne savons pas s'il existe ou pas un algorithme en temps polynomial permettant de répondre à cette question en toute généralité. Néanmoins, on sait que si $\texttt{EADT}_\wedge$ (respectivement, EDT) satisfait **SE**, alors $\texttt{EADT}_\wedge$ (respectivement, EDT) satisfait **EQ**. En effet, nous savons que deux formules propositionnelles Σ_1 et Σ_2 sont logiquement équivalentes si et seulement si nous avons $\Sigma_1 \models \Sigma_2$ et $\Sigma_2 \models \Sigma_1$. Un algorithme pour tester l'équivalence consiste simplement en deux tests d'implication générale.

Pour cette raison, on sait prouver que ADT vérifie le test d'équivalence. La preuve est triviale et vient directement du fait que ADT vérifie le test de l'implication générale. En conséquence, ses sous-ensembles DT et $\texttt{ODT}_<$ offrent aussi le test d'équivalence **EQ**.

L'énumération de modèles (ME)

Tout langage propositionnel qui vérifie le test de cohérence et qui offre le conditionnement permet aussi d'énumérer les modèles en temps polynomial (dans la taille de son entrée et de l'ensemble des modèles à énumérer) [DM02] et même avec un délai polynomial. En effet, soit Σ une formule d'un tel langage. On peut construire en temps polynomial dans la taille de Σ et de l'ensemble de ses modèles, un arbre binaire T dont les chemins de la racine aux feuilles représentent les modèles de Σ.

Cet arbre peut être construit récursivement de sa racine aux feuilles. Chaque chemin des arbres obtenus pendant le processus de construction correspond à un terme cohérent sur les variables de Σ. On

commence par tester si Σ est cohérente (ce qui peut donc être déterminé en temps polynomial). Si Σ est incohérente, alors Σ a un ensemble vide de modèles et il n'y a rien à construire. Sinon on part d'un arbre $T = T_0$ réduit à une feuille (et qui correspond à la conjonction vide de littéraux). On se donne un suivant un ordre quelconque (mais total et strict) des variables $x_1 < x_2 < \cdots < x_n$ sur $Var(\Sigma)$ et en considérant les variables selon cet ordre, on va s'efforcer de prolonger à chaque étape l'arbre courant en ajoutant des nœuds et des arcs lorsque les termes correspondants aux chemins étendus sont cohérents avec Σ. A l'étape i, pour chaque chemin (associé à un terme cohérent γ) de l'arbre courant T_i :

— si $\Sigma \mid (\gamma \wedge x_i)$ est cohérent, on étend dans T_{i+1} la feuille de T_i associée à γ en ajoutant un arc étiqueté par x_i vers une nouvelle feuille ;

— si $\Sigma \mid (\gamma \wedge \neg x_i)$ est cohérent, on étend en outre dans T_{i+1} la feuille de T_i associée à γ en ajoutant un arc étiqueté par $\neg x_i$ vers une nouvelle feuille ;

Comme le langage considéré satisfait le conditionnement et le test de cohérence, chaque étape de traitement d'une feuille peut être réalisée en temps polynomial. De plus, nous savons qu'au moins un des deux tests de cohérence sera satisfait puisque Σ est cohérente. Il existe donc au moins un modèle dans lequel les variables x_i sont soit à vrai soit à faux. Finalement, à la fin des itérations, chaque feuille de l'arbre de décision $T = T_n$ obtenu correspond à un des modèles de Σ (et deux feuilles distinctes à des modèles distincts).

Comme le langage des $\texttt{EADT}_\wedge$ satisfait le test de cohérence et le conditionnement, l'algorithme ci-dessus permet bien d'énumérer les modèles de la formule $\texttt{EADT}_\wedge$ Σ donnée en entrée, en temps polynomial dans la taille de Σ et le nombre de modèles de Σ. Donc le langage $\texttt{EADT}_\wedge$ et ses sous-langages satisfont la requête **ME**.

Synthèse des résultats obtenus sur les requêtes

Nous résumons ici l'ensemble des requêtes définies dans la carte de compilation et nous comparons les nouveaux langages introduits aux principaux langages déjà étudiés et situés dans la carte de compilation. Le tableau 4.1 indique les requêtes offertes par les différents langages.

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
$\texttt{EADT}_\wedge$	√	√	√	√	?	?	√	√
EDT	√	√	√	√	?	?	√	√
ADT	√	√	√	√	√	√	√	√
DT	√	√	√	√	√	√	√	√
$\texttt{ODT}_<$	√	√	√	√	√	√	√	√
d-DNNF	√	√	√	√	?	○	√	√
$\texttt{OBDD}_<$	√	√	√	√	√	√	√	√

TABLE 4.1 – Requêtes. √ signifie "satisfait" et ○ signifie "ne satisfait pas sauf si P = NP."

Notons que le langage ADT des arbres de décision affine et ses sous-ensembles DT et $\texttt{ODT}_<$ satisfont l'ensemble des requêtes proposées dans la carte de compilation. Ainsi, ADT et ses sous-ensembles sont équivalents au langage $\texttt{OBDD}_<$ du point de vue des requêtes offertes. De même, $\texttt{EADT}_\wedge$ et son sous-ensemble EDT sont équivalents au langage d-DNNF pour ce qui est des requêtes pour lesquelles on connaît des algorithmes en temps polynomial pour les réaliser.

4.2.2 Les transformations

Identifions maintenant les transformations offertes par les langages introduits.

Le conditionnement (CD)

Nous commençons par montrer que le langage $\texttt{EADT}_\wedge$ offre la transformation fondamentale qu'est le conditionnement et qu'il en est de même pour ses sous-ensembles ADT, EDT, DT et $\texttt{ODT}_<$. Pour cela, nous montrons que le conditionnement de la clause affine étiquetant un nœud affine n'affecte pas les contraintes supplémentaires utilisées pour définir les sous-langages ADT, EDT, DT et $\texttt{ODT}_<$.

Soient γ un terme cohérent et Σ une formule $\texttt{EADT}_\wedge$. Lorsque nous remplaçons chaque clause affine δ_i (qui étiquette un nœud de décision affine de Σ) par $\delta_i \mid \gamma$, nous obtenons une formule $\texttt{EADT}_\wedge$ équivalente à $\Sigma \mid \gamma$. En effet, le conditionnement est une opération qui se « distribue » sur les connecteurs (il s'agit de remplacer des variables par des constantes booléennes). De plus, il ne remet pas en cause les propriétés du nœud : le conditionnement d'un nœud de décision préserve la nature de ce nœud, le conditionnement d'un nœud de conjonction décomposable voire affine décomposable préserve la décomposabilité de ce nœud et enfin le conditionnement d'une feuille ne la change pas. Ainsi, nous avons :

— $(((\delta \oplus \top) \wedge \Sigma_1) \vee (\delta \wedge \Sigma_2)) \mid \gamma \equiv (((\delta \mid \gamma) \oplus \top) \wedge (\Sigma_1 \mid \gamma)) \vee (((\delta \mid \gamma) \wedge (\Sigma_2 \mid \gamma)))$;

— $(\Sigma_1 \wedge \cdots \wedge \Sigma_k) \mid \gamma \equiv (\Sigma_1 \mid \gamma) \wedge \cdots \wedge (\Sigma_k \mid \gamma)$;

— $\bot \mid \gamma \equiv \bot$ et $\top \mid \gamma \equiv \top$.

La première équivalence dans cette énumération montre que le conditionnement d'un nœud de décision conduit encore à un nœud de décision : quand δ est une clause affine, $\delta \mid \gamma$ est encore une clause affine [3]. La deuxième équivalence montre directement que la décomposabilité des nœuds de conjonction est préservée par le conditionnement : si $\Sigma_1, \ldots, \Sigma_k$ ne partagent aucune variable, il en va de même pour $\Sigma_1 \mid \gamma, \ldots, \Sigma_k \mid \gamma$ puisque le conditionnement d'une formule ne peut qu'éliminer des variables apparaissant dans celle-ci. Pour la même raison, la décomposabilité affine des nœuds de conjonction est préservée elle aussi.

Par ailleurs, on observe aisément que le conditionnement d'une formule $\texttt{EADT}_\wedge$ ne contenant pas de nœuds de conjonction n'introduit pas de tels nœuds. Autrement dit, le conditionnement d'une formule ADT produit encore une formule ADT.

Pour ce qui est du langage EDT, on observe facilement que le conditionnement d'un nœud de décision standard (i.e., un nœud de décision affine unitaire) produit encore un nœud de décision standard : remplacer une variable x par $\top$ ou par $\bot$ ne fait pas croître la taille de la clause affine. Ainsi, le conditionnement d'une formule EDT produit encore une formule EDT.

Une conséquence directe des deux observations précédentes est que le conditionnement d'une formule DT est encore une formule DT. Enfin, lorsque nous considérons une formule DT respectant un ordre $<$ sur les variables, son conditionnement respecte lui aussi cet ordre. En d'autres termes, le conditionnement d'une formule $\texttt{ODT}_<$ est encore une formule $\texttt{ODT}_<$.

Le processus de remplacement de variables par des constantes booléennes qu'est le conditionnement (comme décrit par les équivalence ci-dessus, vues comme des règles de réécriture orientées de la gauche vers la droite) s'effectue en temps linéaire en la taille de la formule Σ plus la taille du terme γ. Donc les langages $\texttt{EADT}_\wedge$, ADT, EDT, DT et $\texttt{ODT}_<$ offrent le conditionnement **CD**.

L'oubli (FO)

Nous montrons qu'aucun des cinq langages $\texttt{EADT}_\wedge$, ADT, EDT, DT et $\texttt{ODT}_<$ ne vérifie l'oubli sauf si P = NP. Pour cela, nous mettons en évidence une réduction polynomiale depuis le test de validité pour DNF vers l'oubli pour $\texttt{ODT}_<$. Soit la formule DNF $\Sigma = \bigvee_{i=1}^{m} \gamma_i$ où chaque γ_i ($i \in \{1, \ldots, m\}$) est un terme. On associe en temps polynomial suivant la taille de Σ la formule DNF déterministe Σ' définie par $\bigvee_{i=1}^{m}(\gamma_i \wedge n_i \wedge \bigwedge_{j=1}^{i-1} \neg n_j)$ où les n_i ($i \in \{1, \ldots, m\}$) sont de nouvelles variables propositionnelles qui

3. Si $\delta \mid \gamma$ équivaut à la constante $\bot$ (respectivement $\top$), on peut remplacer le nœud par son fils gauche (respectivement droit).

n'apparaissant pas dans Σ. Ces nouvelles variables sont introduites pour assurer le déterminisme de Σ'. Par construction Σ' est donc une formule DNF déterministe (ses termes sont deux à deux conjointement incohérents). De plus, nous avons $\Sigma \equiv \exists n_1, \ldots, n_m.\Sigma'$. Nous pouvons maintenant associer en temps polynomial à Σ' une formule $\mathtt{ODT}_<$ équivalente Σ'', telle que

$$\Sigma'' = \langle n_1, \langle n_2, \ldots, \langle n_{m-1}, \langle n_m, \bot, \gamma'_m\rangle, \gamma'_{m-1}\rangle \ldots, \gamma'_1\rangle$$

où chaque γ'_i ($i \in \{1, \ldots, m\}$) est une formule $\mathtt{ODT}_<$ équivalente au terme γ_i. La formule Σ'' ainsi définie est schématisée à la figure 4.1.

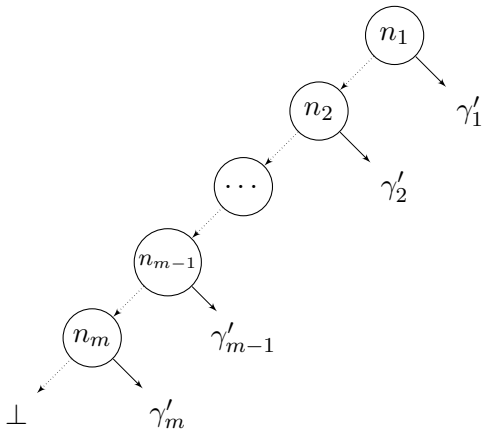


FIGURE 4.1 – Arbre de décision ordonné Σ''.

Quel que soit l'ordre $<$ des variables de $Var(\Sigma)$ considéré, la taille de γ'_i est linéaire en la taille du terme γ_i. Enfin, Σ''' respecte l'extension de l'ordre $<$ sur les variables de $Var(\Sigma) \cup \{n_1, \ldots, n_m\}$ défini par $\forall i \in \{1, \ldots, m-1\}$, $n_i < n_{i+1}$ et $\forall i \in \{1, \ldots, m\}$, $\forall x \in Var(\Sigma), n_i < x$.

Comme $\Sigma' \equiv \Sigma''$, nous avons donc $\Sigma \equiv \exists n_1, \ldots, n_m.\Sigma''$. Supposons que le langage $\mathtt{ODT}_<$ (respectivement DT, EDT, ADT et $\mathtt{EADT}_\wedge$) vérifie l'oubli **FO**. Nous pourrions alors calculer en temps polynomial dans la taille de Σ'' une formule $\mathtt{ODT}_<$ (et donc une formule DT, ADT, EDT et $\mathtt{EADT}_\wedge$) $\exists n_1, \ldots, n_m.\Sigma''$, et donc obtenir ainsi une formule $\mathtt{ODT}_<$ logiquement équivalente à Σ. Comme chacun des langages $\mathtt{ODT}_<$, DT, ADT, EDT et $\mathtt{EADT}_\wedge$ vérifie **VA**, nous obtiendrions un algorithme en temps polynomial en la taille de Σ permettant de déterminer si Σ est valide. Or DNF ne satisfait pas **VA** sauf si P=NP. Donc les langages $\mathtt{ODT}_<$, DT, ADT, EDT et $\mathtt{EADT}_\wedge$ ne satisfont pas **FO** sauf si P=NP.

L'oubli d'une variable (SFO)

Nous montrons que les langages des ADT, DT et $\mathtt{ODT}_<$ permettent l'oubli d'une variable. Cela est, en fait, une conséquence directe du fait que ces langages vérifient le conditionnement **CD** et la disjonction bornée $\vee\mathbf{BC}$. En effet, soit Σ une formule propositionnelle et x une variable propositionnelle. Nous avons $\exists x.\alpha \equiv (\Sigma \mid \neg x) \vee (\Sigma \mid x)$. Donc tout langage permettant le conditionnement et la disjonction bornée offre l'oubli d'une variable. Comme ADT et ses sous-langages permettent le conditionnement et la disjonction bornée, ils offrent aussi **SFO**.

Pour ce qui est des langages EDT (respectivement $\mathtt{EADT}_\wedge$), la question reste ouverte. On peut néanmoins montrer pour chacun d'eux qu'il satisfait **SFO** si et seulement si il satisfait la disjonction bornée $\vee\mathbf{BC}$ (une question qui reste également ouverte). En effet, en tenant le même raisonnement que

ci-dessus, comme **CD** est satisfait, si $\vee\mathbf{BC}$ l'est aussi, alors **SFO** l'est également. Réciproquement, considérons deux formules EDT (respectivement $\text{EADT}_\wedge$) Σ_1 et Σ_2, et x une nouvelle variable ($x \notin Var(\Sigma_1) \cup Var(\Sigma_2)$). La formule EDT (respectivement $\text{EADT}_\wedge$) $\Sigma = \langle x, \Sigma_1, \Sigma_2 \rangle$ peut être construite en temps linéaire en la taille de Σ_1 et de Σ_2. De plus, par construction, nous avons $\Sigma_1 \vee \Sigma_2 \equiv \exists x.\Sigma$. Si le langage EDT (respectivement $\text{EADT}_\wedge$) satisfaisait **SFO**, nous serions en mesure de calculer en temps polynomial en la taille de Σ_1 et de Σ_2 une formule EDT équivalente à $\Sigma_1 \vee \Sigma_2$. Cela se ferait simplement en oubliant x dans la formule Σ. Donc si **SFO** est satisfait par EDT (respectivement $\text{EADT}_\wedge$), alors $\vee\mathbf{BC}$ est satisfait par EDT (respectivement $\text{EADT}_\wedge$).

La conjonction non bornée ($\wedge\mathbf{C}$)

Nous montrons qu'aucun des langages $\text{ODT}_<$, DT, ADT, EDT et $\text{EADT}_\wedge$ ne permet la conjonction non bornée $\wedge\mathbf{C}$ en temps polynomial. Pour cela, nous mettons en évidence une réduction polynomiale depuis le test de cohérence pour le langage CNF. Soit une formule CNF $\Sigma = \bigwedge_{i=1}^{m} \delta_i$ où chaque $\delta_i (i \in 1, \ldots, m)$ est une clause. Nous pouvons associer à chaque δ_i une formule δ'_i sous forme $\text{ODT}_<$ logiquement équivalente à δ_i en temps linéaire en la taille de δ_i.

Supposons que le langage des arbres de décision ordonnés ($\text{ODT}_<$) (respectivement DT, ADT, EDT et $\text{EADT}_\wedge$) vérifie la conjonction non bornée ($\wedge\mathbf{C}$). Dans ce cas, nous pourrions transformer une formule propositionnelle Σ en une formule $\text{ODT}_<$ Σ' (et donc en une formule DT, ADT, EDT et $\text{EADT}_\wedge$), que nous calculerions en temps polynomial dans la taille de Σ. En effet, pour cela, nous calculerions Σ', une formule $\text{ODT}_<$ logiquement équivalente à $\bigwedge_{i=1}^{m} \delta'_i$, et donc logiquement équivalente à Σ. Comme chacun des langages $\text{ODT}_<$, DT, ADT, EDT et $\text{EADT}_\wedge$ vérifie le test de cohérence **CO**, nous aurions alors un algorithme en temps polynomial en la taille de Σ qui permettrait de décider sa cohérence. En conséquence, nous aurions montré que P = NP.

Donc le langage des arbres de décision ordonnés ($\text{ODT}_<$) et ses sur-ensembles DT, ADT, EDT et $\text{EADT}_\wedge$ ne permettent pas la conjonction non bornée $\wedge\mathbf{C}$ sauf si P = NP.

La conjonction bornée ($\wedge\mathbf{BC}$)

Dans un premier temps, nous montrons que les langages EDT et $\text{EADT}_\wedge$ ne vérifient pas $\wedge\mathbf{BC}$ sauf si P = NP. Ensuite, nous montrons que les langages ADT et DT permettent $\wedge\mathbf{BC}$. Enfin, nous montrons que le langage $\text{ODT}_<$ offre aussi la transformation $\wedge\mathbf{BC}$.

Pour démontrer que les langages EDT et $\text{EADT}_\wedge$ n'offrent pas $\wedge\mathbf{BC}$ sauf si P = NP, nous mettons en évidence une réduction depuis le test de cohérence pour CNF. Considérons une formule CNF $\Sigma = \bigwedge_{i=1}^{m} \delta_i$, portant sur les variables $Var(\Sigma) = \{x_1, \ldots, x_n\}$, où chaque formule δ_i ($i \in \{1, \ldots, m\}$) est une clause. Nous pouvons associer en temps polynomial à la formule Σ deux formules CNF Σ_1 et Σ_2 telles que :

- $\Sigma_1 = \bigwedge_{i=1}^{m} \delta_i^i$, où chaque $\delta_i^i (i \in \{1, \ldots, m\})$ est la clause δ_i dans laquelle chaque variable x_j ($j \in \{1, \ldots, n\}$) est remplacée par une nouvelle variable x_j^i ;
- $\Sigma_2 = \bigwedge_{j=1}^{n} (\bigwedge_{i=1}^{m} ((\neg x_j \vee x_j^i) \wedge (x_j \vee \neg x_j^i)))$.

Notons que chaque variable de Σ_1 n'apparait qu'une fois au maximum, dans une seule clause. Σ_2 relie les variables de Σ aux nouvelles variables créées (leurs "clones"). Ainsi, Σ_2 est équivalent à $\bigwedge_{j=1}^{n} (\bigwedge_{i=1}^{m} (x_j \Leftrightarrow x_j^i))$. Par construction, la formule Σ est cohérente si et seulement si $\Sigma_1 \wedge \Sigma_2$ est cohérente. On sait aussi associer en temps polynomial aux formules Σ_1 et Σ_2 des formules EDT (et donc des formules $\text{EADT}_\wedge$) équivalentes Σ'_1 et Σ'_2 (respectivement). En effet, par construction, les connecteurs $\bigwedge$ les plus à l'extérieur de Σ_1 ($\bigwedge_{i=1}^{m}$) et de Σ_2 ($\bigwedge_{j=1}^{n}$) sont décomposables. Par conséquent, la formule $\Sigma'_1 = \bigwedge_{i=1}^{m} \delta'^i_i$ où chaque δ'^i_i ($i \in \{1, \ldots, m\}$) est une représentation de la clause δ_i^i en formule DT, est bien une formule EDT équivalente à Σ_1. De plus, la formule $\Sigma'_2 = \bigwedge_{j=1}^{n} \alpha_j$ où chaque α_j ($j \in 1\{1, \ldots, n\}$) est

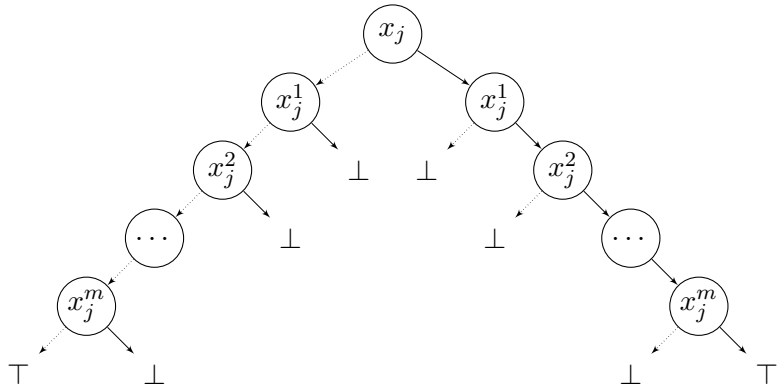


FIGURE 4.2 – Arbre de décision α_j.

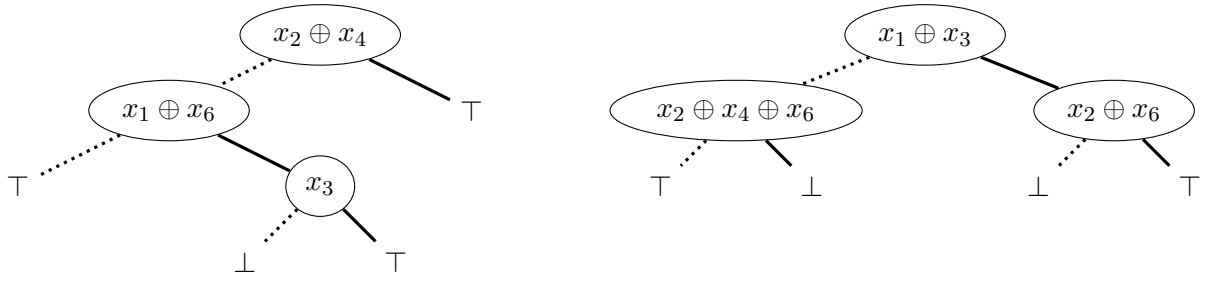
la représentation en formule DT de $\bigwedge_{i=1}^{m}((\neg x_j \vee x_j^i) \wedge (x_j \vee \neg x_j^i))$. Cette formule est équivalente à la formule DNF $(x_j \wedge \bigwedge_{i=1}^{m} x_j^i) \vee (\neg x_j \wedge \bigwedge_{i=1}^{m} \neg x_j^i)$. Donc chaque α_j ($j \in \{1, \ldots, n\}$) peut être vu comme la formule DT suivante, schématisée à la figure 4.2, qui peut être calculée en temps linéaire à partir de $\bigwedge_{i=1}^{m}(x_j \Leftrightarrow x_j^i)$:

$$\langle x_j, \langle x_j^1, \langle x_j^2, \ldots, \langle x_j^m, \top, \bot\rangle, \bot\rangle, \bot\rangle \ldots\rangle, \langle x_j^1, \bot, \langle x_j^2, \bot, \langle \ldots, \langle x_j^m, \bot, \top\rangle, \ldots\rangle$$

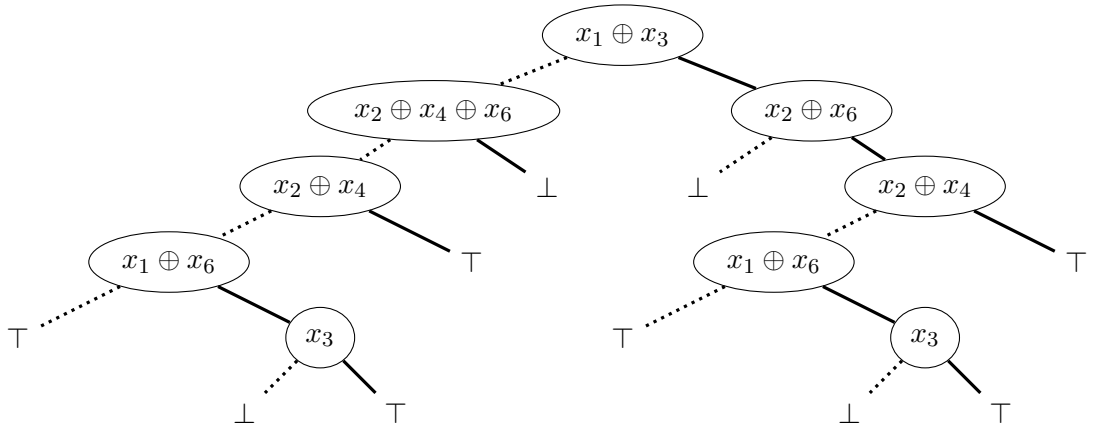
Finalement, si l'un des langages EDT ou $\text{EADT}_\wedge$ vérifiait la conjonction bornée $\wedge\mathbf{BC}$, alors pour tester la cohérence d'une formule CNF Σ, il suffirait de générer d'abord les formules Σ_1' et Σ_2', calculables en temps polynomial en la taille de Σ. Il faudrait ensuite calculer une formule EDT (donc $\text{EADT}_\wedge$) équivalente $\Sigma_1' \wedge \Sigma_2'$ en temps polynomial (ce qui serait possible sous les hypothèses faites) et finalement tester la cohérence de cette formule, ce qui se vérifie en temps polynomial puisque EDT (et $\text{EADT}_\wedge$) offre **CO**. Nous aurions donc au final conçu un algorithme en temps polynomial pour déterminer la cohérence d'une formule CNF. Cela entrainerait que P = NP. En conclusion, le langage des arbres de décision étendus (EDT) et son sur-ensemble $\text{EADT}_\wedge$ n'offrent donc pas $\wedge\mathbf{BC}$ sauf si P = NP.

A contrario, les langages ADT et DT vérifient $\wedge\mathbf{BC}$. En effet, soient Σ_1 et Σ_2 deux formules ADT (respectivement, DT). Il est possible de construire une formule ADT (respectivement, DT) Σ logiquement équivalente à la formule $\Sigma_1 \wedge \Sigma_2$ en remplaçant tous les nœuds étiquetés par $\top$ dans la formule Σ_2 par une copie de la formule Σ_1. Il suffit donc de parcourir la formule Σ_2 et de remplacer chaque feuille $\top$ que nous trouvons par une copie de Σ_1. Ceci est réalisable en temps quadratique en la taille de Σ_1 plus la taille de Σ_2.

Exemple 63 *Soient* Σ_1 *et* Σ_2 *les deux formules* ADT *suivantes :*

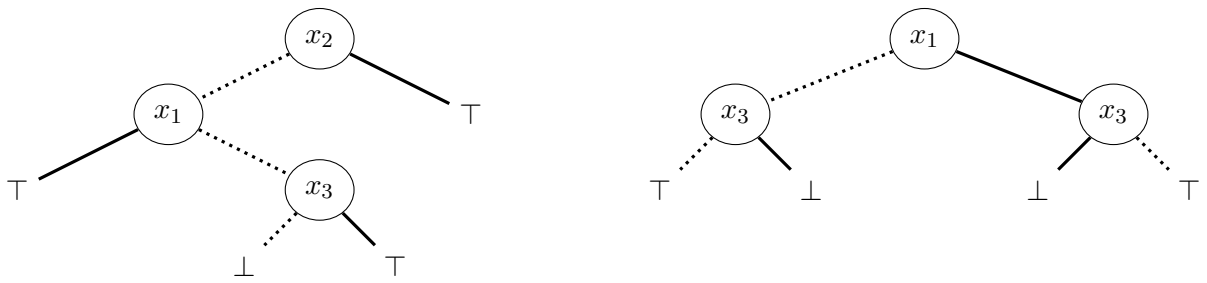


La conjonction de ces deux formules $\Sigma_1 \wedge \Sigma_2$ *est équivalente à la formule* ADT *:*

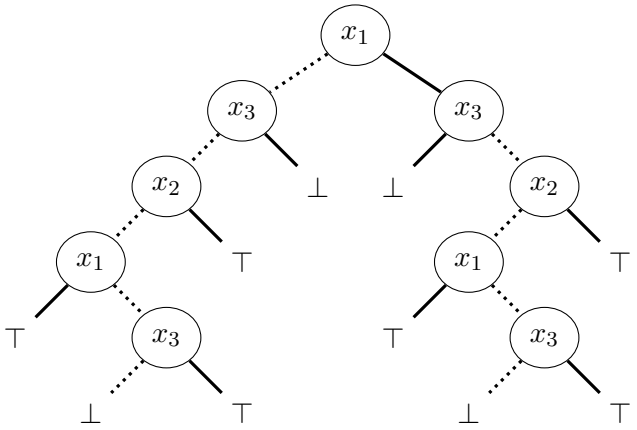


Pour le langage $\texttt{ODT}_<$, la preuve est similaire à la preuve précédente. Considérons une formule Σ équivalente à $\Sigma_1 \wedge \Sigma_2$, obtenue en remplaçant tout nœud étiqueté par $\top$ dans Σ_1 par une copie de Σ_2. En général, une telle formule $\texttt{DT}\ \Sigma$ n'est pas une formule $\texttt{ODT}_<$ puisque certains chemins depuis la racine de Σ à des feuilles peuvent contenir deux nœuds de décision étiquetés de la même variable de décision. Dans ce cas, il suffit d'appliquer à Σ la méthode de réduction déjà présentée à la section 4.2. Il s'agit de propager de la racine vers les feuilles les valeurs de vérité des variables de décision : si $N = \langle x, N_-, N_+ \rangle$ est un nœud de décision de Σ alors toute occurrence de x dans la sous-formule de Σ enracinée en N_- (respectivement, N_+) est remplacée par $\bot$ (respectivement, $\top$). Cela revient à remplacer tout nœud successeur $N' = \langle x, N'_-, N'_+ \rangle$ de N par N'_- si le chemin de N à N' passe par N_- et par N'_+ si le chemin de N à N' passe par N_+. Une fois ce traitement réalisé, on est sûrs qu'aucune occurrence de x n'existe dans toute formule enracinée à un nœud qui est un descendant d'un nœud de décision portant sur x. Pour cette raison, la formule $\texttt{DT}$ ainsi construite respecte l'ordre $<$ et est donc une formule $\texttt{ODT}_<$.

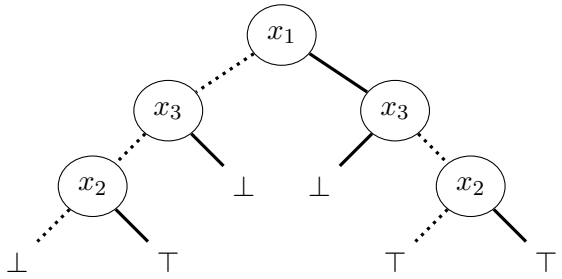
Exemple 64 *Soient les deux formules* $\texttt{ODT}_<$ Σ_1 *et* Σ_2 *suivantes :*



Après conjonction de Σ_1 *et* Σ_2 *nous obtenons la formule* $\texttt{DT}\ \Sigma$ *suivante :*



Le réduction de Σ donne la formule $\texttt{ODT}_<$ *suivante :*



Le langage $\texttt{ODT}_<$ permet donc la conjonction bornée en temps polynomial.

La disjonction non bornée ($\vee\mathbf{C}$)

Nous montrons que les langages $\texttt{ODT}_<$, DT, ADT, EDT et $\texttt{EADT}_\wedge$ n'offrent pas la disjonction non bornée sauf si P = NP. Nous avons vu que tout terme de TE peut être traduit en temps polynomial en une formule $\texttt{ODT}_<$ équivalente. Si l'un de ces langages vérifiant $\vee\mathbf{C}$, alors on pourrait construire dans le même langage en temps polynomial une représentation d'une disjonction finie quelconque de termes, donc on aurait une traduction en temps polynomial de DNF vers le langage en question. Comme chacun de ces langages offre **VA**, on saurait alors décider en temps polynomial si une formule DNF quelconque est valide, ce qui entraînerait P = NP. Par conséquent, les langages $\texttt{ODT}_<$, DT, ADT, EDT et $\texttt{EADT}_\wedge$ n'offrent pas la disjonction non bornée $\vee\mathbf{C}$ sauf si P = NP.

La disjonction bornée ($\vee\mathbf{BC}$)

Nous montrons que les langages $\texttt{ODT}_<$, DT et ADT permettent la disjonction bornée. Cela est une conséquence directe du fait qu'ils offrent la conjonction bornée et la négation. En suivant les lois de De Morgan, pour calculer la disjonction de deux formules ADT (respectivement, $\texttt{ODT}_<$, DT) Σ_1 et Σ_2, il suffit de calculer : $\neg(\neg\Sigma_1 \vee \neg\Sigma_2)$. Le résultat est une formule ADT (respectivement, $\texttt{ODT}_<$, DT).

Pour les langages EDT et $\texttt{EADT}_\wedge$, la question reste ouverte.

la négation ($\neg\mathbf{C}$)

Nous montrons que les langages $\texttt{ODT}_<$, DT, et ADT offrent la négation $\neg\mathbf{C}$.

Pour cela, nous expliquons comment il est possible d'associer en temps linéaire, via une fonction de traduction neg, à n'importe quelle formule ADT Σ une formule ADT $neg(\Sigma)$ logiquement équivalente à

$\neg\Sigma$. La fonction neg prend en paramètre un nœud N d'une formule ADT Σ. L'algorithme de traduction commence avec $N = R_\Sigma$. La traduction neg est définie récursivement comme :

— si $N = \langle \delta, N_-, N_+ \rangle$, alors $neg(N) = \langle \delta, neg(N_-), neg(N_+) \rangle$;
— si $N = \top$, alors $neg(N) = \bot$;
— si $N = \bot$, alors $neg(N) = \top$.

En d'autres termes, pour calculer $neg(\Sigma)$, il suffit de traverser Σ jusqu'à ses feuilles et de remplacer chaque feuille $\top$ par une feuille $\bot$ et chaque feuille $\bot$ par une feuille $\top$. Le fait que $neg(\Sigma)$ soit équivalent à $\neg\Sigma$ se prouve par induction structurelle sur Σ. Le cas de base concerne les feuilles $\top$ et $\bot$ et est trivial. Reste donc à considérer le cas des nœuds internes N de la forme $N = \langle \delta, N_-, N_+ \rangle$. Par définition, la formule Σ_N enracinée en N équivaut à $(\delta \oplus \top) \wedge \Sigma_{N_-}) \vee (\delta \wedge \Sigma_{N_+})$, où Σ_{N_-} (respectivement, Σ_{N_+}) est la formule enracinée en N_- (respectivement, N_+). Cela conduit directement à l'équivalence $\neg\Sigma \equiv (\delta \vee \neg\Sigma_{N_-}) \wedge ((\delta \oplus \top) \vee \neg\Sigma_{N_+})$. Après distribution du $\wedge$ sur les $\vee$ et simplification, nous obtenons la formule (toujours équivalente à $\neg\Sigma$) : $((\delta \oplus \top) \wedge \neg\Sigma_{N_-}) \vee (\delta \wedge \neg\Sigma_{N_+})$. Par hypothèse d'induction, $\neg\Sigma_{N_-}$ équivaut à $neg(N_-)$ et $\neg\Sigma_{N_+}$ équivaut à $neg(N_+)$. Ainsi, on à l'équivalence $\neg\Sigma \equiv ((\delta \oplus \top) \wedge neg(N_-)) \vee (\delta \wedge neg(N_+))$. Or, par définition, $((\delta \oplus \top) \wedge neg(N_-)) \vee (\delta \wedge neg(N_+))$ équivaut à la formule enracinée en $\langle \delta, neg(N_-), neg(N_+) \rangle$, donc à la formule enracinée en $neg(N)$ et cela termine la preuve par induction.

Clairement, neg préserve la nature des nœuds de décision (si $N = \langle \delta, N_-, N_+ \rangle$ est un nœud de décision affine unitaire alors $neg(N) = \langle \delta, neg(N_-), neg(N_+) \rangle$ est aussi un nœud de décision affine unitaire), ce qui implique que quand Σ est une formule DT, alors $neg(\Sigma)$ est encore une formule DT. neg préserve aussi l'ordre des décisions sur chaque chemin, ce qui implique que si Σ est une formule $\text{ODT}_<$, alors $neg(\Sigma)$ est encore une formule $\text{ODT}_<$.

La question de déterminer si EDT et $\text{EADT}_\wedge$ satisfont $\neg\mathbf{C}$ reste ouverte.

Synthèse des résultats sur les transformations

Nous résumons dans le tableau 4.2 suivant les transformations offertes par les nouveaux langages introduits. Nous remarquons qu'à l'instar des requêtes, les langages $\text{ODT}_<$, DT et ADT sont équivalents au langage $\text{OBDD}_<$ du point de vue des transformations qu'ils offrent. De même, $\text{EADT}_\wedge$ et son sous-ensemble EDT sont équivalents au langage d-DNNF pour ce qui est des transformations pour lesquelles on connaît des algorithmes en temps polynomial pour les réaliser.

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
$\text{EADT}_\wedge$	$\surd$	○	?	○	○	○	?	?
EDT	$\surd$	○	?	○	○	○	?	?
ADT	$\surd$	○	$\surd$	○	$\surd$	○	$\surd$	$\surd$
DT	$\surd$	○	$\surd$	○	$\surd$	○	$\surd$	$\surd$
$\text{ODT}_<$	$\surd$	○	$\surd$	○	$\surd$	○	$\surd$	$\surd$
d-DNNF	$\surd$	○	○	○	○	○	○	•
$\text{OBDD}_<$	$\surd$	•	$\surd$	•	$\surd$	•	$\surd$	$\surd$

TABLE 4.2 – Transformations. $\surd$ signifie "satisfait", • signifie "ne satisfait pas" et ○ signifie "ne satisfait pas sauf si P = NP".

4.2.3 Efficacité spatiale

Dans cette section, nous présentons les différents résultats que nous avons obtenus concernant l'efficacité spatiale des nouveaux langages introduits, relativement à plusieurs langages cibles déjà insérés la carte de compilation. Il s'agit des langages SDNNF, $\texttt{d-DNNF}_\texttt{T}$, d-DNNF et $\texttt{OBDD}_<$. Le langage $\texttt{d-DNNF}_\texttt{T}$ est le sous-ensemble de d-DNNF introduit par Pipatsrisawat et Darwiche [PD08], qui contient les formules d-DNNF qui respectent un arbre donné T de décomposition des variables. sd-DNNF est l'union de tous ces langages quand T varie. Chaque langage $\texttt{OBDD}_<$ est un sous-ensemble de sd-DNNF.

La figure suivante 4.3 représente le graphe d'inclusion des différents langages considérés :

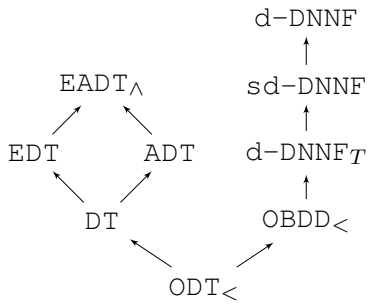


FIGURE 4.3 – Graphe d'inclusion. $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ indique que $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

Étant donné ce graphe d'inclusion et du fait que DT (respectivement EDT) ne satisfait pas plus de requêtes ou de transformations que ADT (respectivement $\texttt{EADT}_\wedge$), nous pouvons déduire que DT (respectivement, EDT) ne peut pas être un meilleur choix que ADT (respectivement $\texttt{EADT}_\wedge$) du point de vue de la carte de compilation (étant données les requêtes et transformations prises en compte). C'est pourquoi nous avons choisi de nous concentrer sur les langages ADT et $\texttt{EADT}_\wedge$.

Dans un premier temps, nous montrons que le langage $\texttt{OBDD}_<$ n'est pas au moins aussi succinct que ADT. Il a été prouvé que la fonction de décalage circulaire de bits cbs_m ne peut pas être représentée en taille polynomiale dans le langage sd-DNNF [PD10]. Comme nous savons que $\texttt{sd-DNNF} \leq_s \texttt{OBDD}_<$ et que $\texttt{sd-DNNF} \leq_s \texttt{d-DNNF}_T$ il n'existe donc pas de représentation en taille polynomiale de cette fonction en $\texttt{OBDD}_<$ ou en $\texttt{d-DNNF}_T$. Nous démontrons que cette (famille de) fonction(s) peut être représentée par une (famille de) formule(s) ADT. Soit m un entier positif. Considérons la fonction booléenne de $2^{m+1} + m$ paramètres $cbs_m(x_1, \ldots, x_{2^m}, y_1, \ldots, y_{2^m}, i_1, \ldots, i_m)$ qui représente la sémantique de la formule CNF α_m suivante :

$$\bigwedge_{i=1}^{2^m} (\neg x_i \vee y_{i+num(i_1,\ldots,i_m) mod 2^m}) \wedge (x_i \vee \neg y_{i+num(i_1,\ldots,i_m) mod 2^m})$$

Dans cette formule, num est une fonction qui prend en entrée les paramètres $\{0,1\}^m$, l'ensemble des nombres entiers naturel représentés sous forme binaire, et donne en retour l'entier représenté par le nombre binaire $i_1, \ldots, i_m$. De cette manière, les variables $i_1, \ldots, i_m$ représentent le nombre précis de bits de la chaîne $x_1 \ldots x_{2^m}$ devant être modifiés. La fonction $cbs_m(x_1, \ldots, x_{2^m}, y_1, \ldots, y_{2^m}, i_1, \ldots, i_m) = 1$ lorsque les variables modifiées $x_1, \ldots, x_{2^m}$ et les variables $y_1, \ldots, y_{2^m}$ sont égales deux à deux modulo le décalage d'indices num.

Nous remarquons que la formule α_m est représentable par une formule affine de taille linéaire en la taille de α_m. En effet, nous avons $\alpha_m \equiv \bigwedge_{i=1}^{2^m} x_i \oplus y_{i+num(i_1,\ldots,i_m) mod 2^m} \oplus \top$. Nous avons donc un algorithme en espace polynomial permettant de représenter la fonction de décalage circulaire de bits en

formule affine. Comme il existe une traduction en temps polynomial depuis le langage AFF des formules affines vers ADT, il existe bien une représentation de chaque α_m dans ADT par une formule de taille polynomiale en la taille de α_m. Par conséquent, le langage $\text{OBDD}_<$ n'est pas au moins aussi succinct que ADT : $\text{OBDD}_< \not\leq_s \text{ADT}$.

Nous montrons maintenant que les langages CNF et DNF ne sont pas au moins aussi succincts que ADT. Pour cela, nous nous basons sur le fait que le langage des formules affine (AFF) est traduisible en temps polynomial en ADT. Cela implique que ADT est au moins aussi succinct que AFF : $\text{ADT} \leq_s \text{AFF}$. Or, le langage CNF n'est pas au moins aussi succinct que AFF ($\text{CNF} \not\leq_s \text{AFF}$) et DNF n'est pas au moins aussi succinct que AFF ($\text{DNF} \not\leq_s \text{AFF}$). En effet, la fonction parité $\oplus_{i=1}^{n} x_i$, qui est une clause affine, ne possède pas de représentation en taille polynomiale en n sous forme CNF ou DNF. Donc les langages CNF et DNF ne sont pas au moins aussi succincts que le langage des arbres de décision affine (ADT) : $\text{CNF} \not\leq_s \text{ADT}$ et $\text{DNF} \not\leq_s \text{ADT}$.

Finalement, nous montrons que le langage $\text{EADT}_\wedge$ est strictement plus succinct que ADT ($\text{EADT}_\wedge <_s \text{ADT}$). Comme le langage ADT est inclus dans $\text{EADT}_\wedge$, nous avons de façon évidente que $\text{EADT}_\wedge$ est au moins aussi succinct que ADT ($\text{EADT}_\wedge \leq_s \text{ADT}$). Nous prouvons maintenant que ADT n'est pas au moins aussi succinct que $\text{EADT}_\wedge$: $\text{ADT} \not\leq_s \text{EADT}_\wedge$. Considérons la (famille de) formule(s) CNF $\Sigma_n = \bigwedge_{i=1}^{n}(\neg x_i \vee \neg y_i)$. Il a été montré que cette formule ne peut pas être représentée par une formule AFF[$\vee$] de taille polynomiale en n [FM14]. Or, le langage ADT est traduisible en temps polynomial dans AFF[$\vee$]. Soit α une formule ADT. Soit β la disjonction, pour tous les chemins de α entre sa racine et ses feuilles $\top$ de toutes les conjonctions de clauses affines rencontrées dans le chemin considéré. La formule β peut être calculée en temps quadratique en la taille de α. Par construction, β est une disjonction de formules affines logiquement équivalente à α. Nous avons donc un algorithme en temps polynomial permettant de traduire toute formule ADT en une formule AFF[$\vee$] logiquement équivalente. Par conséquent, la formule CNF Σ_n ne peut pas être représentée par une formule ADT en taille polynomiale en n, sinon nous aurions un algorithme en temps polynomial pour traduire cette CNF en AFF[$\vee$]. À l'inverse, la CNF Σ_n possède une représentation en formule $\text{EADT}_\wedge$ (ainsi qu'une représentation en EDT) de taille polynomiale en n. En effet, comme les clauses $\neg x_i \vee \neg y_i$ pour $i \in \{1, \ldots, n\}$ de Σ_n n'ont aucune variable en commun, et que nous savons qu'une clause est traduisible en temps polynomial en DT, Σ_n peut facilement être traduite en $\text{EADT}_\wedge$ (et en EDT). Il suffit pour cela de traduire chacune des clauses en DT puis de les réunir dans un nœud $\wedge$ décomposable qui comportera donc n fils, chacun représentant une clause de Σ_n. Nous avons donc un algorithme permettant de traduire Σ_n en une formule $\text{EADT}_\wedge$ de taille polynomiale en n. Ce qui implique que ADT n'est pas au moins aussi succinct que $\text{EADT}_\wedge$: $\text{ADT} \not\leq_s \text{EADT}_\wedge$. Comme $\text{EADT}_\wedge \leq_s \text{ADT}$ est vérifié, on obtient bien la conclusion attendue : $\text{EADT}_\wedge <_s \text{ADT}$. Le langage des arbres de décision affine étendus ($\text{EADT}_\wedge$) est ainsi strictement plus succinct que celui des arbres de décision affine (ADT).

En résumé, nous avons obtenu les résultats d'efficacité spatiale suivants :

Proposition 5

- $\text{CNF} \not\leq_s \text{ADT}$
- $\text{DNF} \not\leq_s \text{ADT}$
- $\text{OBDD}_< \not\leq_s \text{ADT}$
- $\text{d-DNNF}_T \not\leq_s \text{ADT}$
- $\text{EADT}_\wedge <_s \text{ADT}$

Sur la base de ces résultats, on peut conclure que $\text{OBDD}_<$ ne domine pas ADT du point de vue de la carte de compilation. De plus, comme $\text{ADT} \subseteq \text{EADT}_\wedge$, $\text{OBDD}_<$ ne domine pas $\text{EADT}_\wedge$. Nos résultats de concision montrent qu'aucun des langages « plats » CNF et DNF n'est au moins aussi succinct que ADT

ou EADT$_\wedge$. Bien que nous ne connaissions pas comment d−DNNF et EADT$_\wedge$ se comparent du point de vue de l'efficacité spatiale, nous pouvons néanmoins conclure que la sous-classe sd−DNNF de d−DNNF ne domine ni ADT, ni EADT$_\wedge$.

4.3 Un compilateur CNF-vers-EADT$_\wedge$

Dans cette section, nous présentons un algorithme de compilation pour traduire le langage CNF vers le langage EADT$_\wedge$. Tout d'abord, nous présentons une version généralisée de la décomposition de Shannon (utilisée pour compiler des formules en Decision−DNNF, en OBDD$_<$, etc.) adaptée à la construction de formules EADT$_\wedge$. Nous présentons ensuite le compilateur que nous avons développé. Enfin, nous présentons les expérimentations que nous avons effectuées avec ce compilateur.

4.3.1 Décompositions de Shannon

Dans un premier temps, nous introduisons un nouveau langage, le langage des formules propositionnelles sous forme normale conjonctive étendue (ECNF). Partant d'une formule CNF (voire directement d'une formule ECNF), les formules considérées par notre algorithme de compilation sont des formules ECNF. Nous introduisons d'abord la notion de clause étendue :

Définition 62 (Clause étendue)
Une clause étendue est une disjonction finie de clauses affines.

Définition 63 (Forme normale conjonctive étendue)
*Le langage des formules sous forme normale conjonctive (*ECNF*) est l'ensemble de toutes les conjonctions finies de clauses étendues.*

Exemple 65 *Formule* ECNF *La formule* $ce = x_1 \vee (x_2 \oplus x_3 \oplus \top) \vee (x_1 \oplus x_3)$ *est une clause étendue.*
La formule $\Sigma = (x_1 \vee x_6) \wedge (x_2 \oplus x_4 \oplus \top) \wedge (x_3 \vee (x_4 \oplus x_5) \vee \neg x_1)$ *est une formule* ECNF.

Notons que la conversion d'une formule CNF en ECNF s'effectue en temps linéaire de manière évidente. En effet, toute clause d'une CNF est une disjonction finie de clauses affines unitaires. Donc toute clause est une clause étendue et par extension, toute formule CNF est une formule ECNF particulière.

Une approche naturelle pour compiler une formule CNF en EADT$_\wedge$ repose sur l'exploitation d'une forme généralisée de la décomposition de Shannon. Cette forme généralisée de la décomposition de Shannon permet d'introduire des nœuds de décision étiquetés non plus par des variables mais par des formules propositionnelles. Nous définissons la décomposition de Shannon généralisée de la manière suivante :

Définition 64 (Expansion de Shannon généralisée)
Pour toutes formules propositionnelles Σ *et* δ*, l'expansion de Shannon généralisée de* Σ *selon* δ *est définie par :*

$$\Sigma \equiv (\delta \wedge \Sigma) \vee (\neg\delta \wedge \Sigma)$$

Comme la décomposition de Shannon classique, le principe de la décomposition de Shannon généralisée est de séparer les modèles possibles de Σ en deux sous-ensembles disjoints : ceux qui satisfont δ et ceux qui satisfont $\neg\delta$. Le nœud $\vee$ introduit dans $(\delta \wedge \Sigma) \vee (\neg\delta \wedge \Sigma)$ est un nœud de décomposition de Shannon généralisée. Nous remarquons ainsi que les nœuds de décision affine des langages ADT, EDT et EADT$_\wedge$ sont en fait des nœuds de décomposition de Shannon généralisé dans lesquels Σ est une formule ADT, EDT ou EADT$_\wedge$ et δ est une clause affine.

Cependant, utilisée directement, la décomposition de Shannon généralisée est peu exploitable pour la compilation puisque l'un des avantages de la décomposition de Shannon classique est qu'une variable est éliminée de Σ à chaque nœud. Ceci garantit qu'un algorithme de compilation utilisant cette méthode voie ses exécutions se terminer. En effet, comme au moins une variable est supprimée à chaque itération, après un nombre fini d'itérations, toutes les variables de Σ sont considérées. Nous devons donc étendre la décomposition de Shannon généralisée afin d'assurer que son application élimine à chaque étape au moins une variable dans Σ.

Soient deux formules propositionnelles Σ et δ, et soit x une variable. Nous notons $\Sigma\mid_{x\leftarrow\delta}$ la formule obtenue en remplaçant toute occurrence de x dans la formule Σ par la formule δ. Avec cette notation, nous définissons la décomposition de Shannon généralisée étendue de la façon suivante :

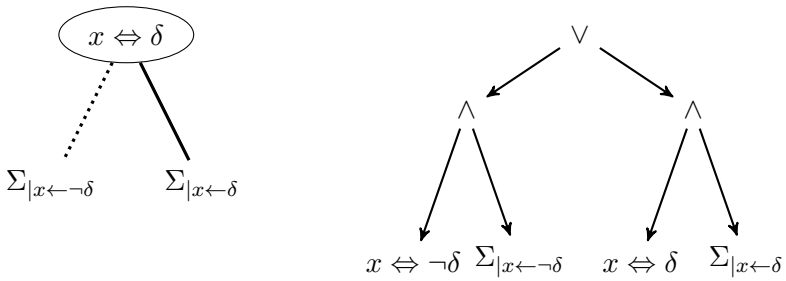
Définition 65 (Décomposition de Shannon généralisée étendue)
Pour toute formule propositionnelle Σ et δ et toute variable x telle que $x \notin Var(\delta)$, la décomposition de Shannon généralisée étendue est définie par :

$$\Sigma \equiv ((x \Leftrightarrow \neg\delta) \wedge \Sigma\mid_{x\leftarrow\neg\delta}) \vee ((x \Leftrightarrow \delta) \wedge \Sigma\mid_{x\leftarrow\delta})$$

Cette équivalence montre qu'en utilisant la décomposition de Shannon généralisée étendue, les modèles de Σ sont séparés en deux sous-ensembles disjoints : ceux qui satisfont $x \Leftrightarrow \delta$ (i.e., ceux où x et δ prennent la même valeur de vérité) et ceux qui satisfont la négation de cette formule, qui équivaut à $x \Leftrightarrow \neg\delta$.

Utiliser la forme généralisée de la décomposition de Shannon étendue consiste à exploiter cette équivalence comme une règle de réécriture, orientée de la gauche vers la droite. On va ainsi remplacer à chaque étape une variable de la formule de départ par une formule propositionnelle ne contenant pas cette variable. La décomposition de Shannon standard est retrouvée dans le cas où la formule $\delta = \top$ est considérée. Effectivement, dans ce cas cela revient à remplacer x par vrai d'un côté et faux de l'autre. La validité de l'expansion de Shannon généralisée étendue vient du fait que la formule Σ est logiquement équivalente à $((x \Leftrightarrow \neg\delta) \wedge \Sigma) \vee ((x \Leftrightarrow \delta) \wedge \Sigma)$, et du fait que pour toute formule propositionnelle δ (ou sa négation), l'expression $(x \Leftrightarrow \delta) \wedge \Sigma$ est logiquement équivalente à $(x \Leftrightarrow \delta) \wedge \Sigma\mid_{x\leftarrow\delta}$.

Exemple 66 *Soit une formule propositionnelle Σ. La décomposition de Shannon généralisée étendue sur la variable x de Σ selon la formule δ est donnée par : $\Sigma \equiv (x \Leftrightarrow \neg\delta \wedge \Sigma_{|x\leftarrow\neg\delta}) \vee (x \Leftrightarrow \delta \wedge \Sigma_{|x\leftarrow\delta})$*



Appliquer cette décomposition revient à créer un nœud de décision étiqueté par la formule $x \Leftrightarrow \delta$, comme montré sur la figure ci-avant. Par construction, dans les formules $\Sigma_{|x\leftarrow\delta}$ et $\Sigma_{|x\leftarrow\neg\delta}$ la variable x n'apparait pas.

Pour construire une formule $\texttt{EADT}_\wedge$ équivalente à une formule $\texttt{ECNF}$ Σ donnée, nous appliquons la décomposition de Shannon généralisée étendue en choisissant pour δ une clause affine qui ne contient pas la variable de décision x.

Algorithme 6 : eadt(Σ)

entrée : une formule ECNF Σ
sortie : une formule EADT$_\wedge$ Σ_{eadt} équivalente à Σ

1 si $\Sigma \equiv \top$ alors return leaf($\top$);
2 si minisat(Σ) $= \bot$ alors return leaf($\bot$);
3 soient $\Sigma_1, \cdots, \Sigma_k$ les composantes connexes de Σ;
4 si $k>1$ *et la décomposition* $\Sigma_1, \cdots, \Sigma_k$ *est affine* alors return aNode(eadt(Σ_1), $\cdots$, eadt(Σ_k));
5 choisir une clause affine simplifiée $x \oplus \delta$ telle que $Var(x \oplus \delta) \subseteq Var(\delta')$ avec δ' une clause de Σ;
6 return dNode($x \oplus \delta$, eadt($\Sigma \mid_{x \leftarrow \delta \oplus \top}$), eadt($\Sigma \mid_{x \leftarrow \delta}$));

Comme $x \Leftrightarrow \delta$ équivaut à $x \oplus \delta \oplus \top$ et $x \Leftrightarrow \neg\delta$ équivaut à $x \oplus \delta$, on peut reformuler la décomposition de Shannon généralisée étendue en utilisant $\oplus$ au lieu de $\Leftrightarrow$:

$$\Delta \equiv ((x \oplus \delta) \wedge \Delta \mid_{x \leftarrow \delta \oplus \top}) \vee ((x \oplus \delta \oplus \top) \wedge \Delta \mid_{x \leftarrow \delta})$$

Par ailleurs, quand δ est une clause affine, $x \oplus \delta \oplus \top$ et $x \oplus \delta$ sont aussi des clauses affines.

De ce fait, sous cette forme, appliquer la décomposition de Shannon généralisée étendue consiste à construire des nœuds de décision affine étiquetés par des clauses de la forme $x \oplus \delta$ et de supprimer à chaque étape la variable x de Σ qui a été sélectionnée.

4.3.2 Algorithme de compilation

L'algorithme 6 décrit en pseudo-code notre compilateur eadt qui se fonde sur la décomposition généralisée étendue de Shannon. Cet algorithme prend en entrée une formule ECNF Σ et retourne une formule EADT$_\wedge$ logiquement équivalente à Σ. Les deux premières lignes gèrent les deux cas particuliers où Σ est valide ou contradictoire. Dans les deux cas, la feuille correspondante ($\top$ si Σ est valide, $\bot$ si Σ est contradictoire) est retournée.

Nous notons au passage que le problème de l'incohérence (respectivement de la validité) pour une formule sous forme ECNF a la même complexité que pour son sous-ensemble CNF, c'est-à-dire qu'il est coNP-complet (respectivement dans P). Cela est évident pour le problème de l'incohérence. En effet, comme toute formule CNF est une formule ECNF, s'il existe un algorithme en temps polynomial vérifiant qu'une formule ECNF est incohérente, nous aurions un algorithme en temps polynomial pour cette requête restreinte aux formules CNF, or ce problème est coNP-complet pour les formules CNF. Pour le problème de validité, une formule ECNF est valide si et seulement si chaque clause étendue de la formule est valide. Une clause étendue $\delta_1 \vee \ldots \vee \delta_k$ (où chaque δ_i, $i \in \{1, \ldots, k\}$ est une clause affine) est valide si est seulement si la formule affine $(\delta_1 \oplus \top) \wedge \ldots \wedge (\delta_k \oplus \top)$ est contradictoire. La négation d'une clause étendue représente donc une formule affine. Or, nous savons que tester si une formule affine est incohérente est faisable en temps polynomial dans sa taille. Donc la validité d'une clause étendue, et au delà d'une ECNF est décidable en temps polynomial en la taille de la formule donnée.

À la ligne 3, on recherche les composantes connexes $\Sigma_1, \cdots, \Sigma_k$ de Σ en parcourant son graphe primal. S'il existe plusieurs composantes et que la conjonction obtenue est affine décomposable[4], les formules Σ_i ($i \in \{1, \ldots, k\}$) sont compilées séparément de manière récursive en formules EADT$_\wedge$. Puis, les formules obtenues sont jointes en utilisant la fonction aNode (ligne 4) par un nœud $\wedge$ qui est alors, par construction, affine décomposable. Notons que la recherche de la décomposition s'effectue avant un branchement. Lorsque Σ ne possède qu'une composante connexe ou que la décomposition n'est pas

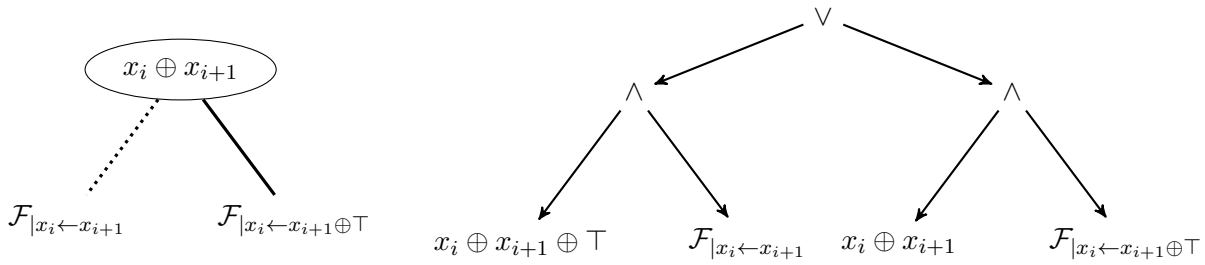
4. On conserve pour cela l'ensemble des clauses affines correspondant aux choix effectués sur la branche courante ; cela n'est pas explicité dans l'algorithme dans un souci de garder celui-ci lisible.

affine, le compilateur choisit une clause affine $x \oplus \delta$ dans laquelle toute variable possède au moins une occurrence dans Σ (ligne 5). Ensuite, nous effectuons le branchement sur cette clause en utilisant la décomposition généralisée étendue de Shannon (ligne 6). L'algorithme est appelé récursivement avec les formules conditionnées, dans lesquels x est remplacé par $\delta \oplus \top$ ou par δ. Enfin, la fonction dNode retourne un nœud de décision correspondant. Nous remarquons que, privé des lignes 3 et 4 le compilateur eadt se réduit à un compilateur de ECNF vers ADT.

La terminaison de l'algorithme eadt est garantie par le fait que par définition, la clause affine simplifiée δ dans $x \oplus \delta$ est une clause affine qui ne contient pas x. Comme ni $\Sigma\ |_{x \leftarrow \delta \oplus \top}$ ni $\Sigma\ |_{x \leftarrow \delta}$ ne contiennent x, les étapes 5 et 6 ne peuvent donc être appliquées qu'un nombre fini de fois. Comme la décomposition généralisée de Shannon exprime une équivalence valide, l'algorithme eadt est correct (la décomposabilité affine des nœuds $\wedge$ éventuellement introduits est garantie par construction).

L'algorithme eadt a été implémenté sur la base du solveur état de l'art MiniSat [ES03], qui s'appuie sur une architecture CDCL (*Conflict Driven Clause Learning*) [MMZ+01b]. Nous avons étendu MiniSat aux formules ECNF. Pour cela, nous avons modifié la façon dont sont gérés les watched littéraux. Nous avons adapté la propagation aux clauses étendues et la gestion des conflits pour prendre en compte des clauses affines.

Afin de pouvoir utiliser eadt il reste à préciser les choix effectués à la ligne 5 : quelle variable x sélectionner, quelle clause affine δ retenir. Une heuristique est indispensable pour cela et le choix de cette heuristique peut conditionner fortement la performance du compilateur (en particulier, avoir un impact important sur la taille de la formule $\text{EADT}_\wedge$ qui sera engendrée). Une heuristique naïve est la suivante (souvent appelée « heuristique lexicographique ») : on ordonne d'abord les variables de la formule d'entrée Σ selon un ordre particulier $x_1 < \ldots < x_n$ et à l'étape i sur la branche courante, on choisit $x = x_i$ et $\delta = x_{i+1}$. Puis on remplace toutes les occurrences de x_i dans la formule Σ par $x_{i+1} \oplus \top$ d'un côté, que nous notons $\Sigma[x_i \leftarrow x_{i+1} \oplus \top]$, et par x_{i+1} de l'autre, que nous notons $\Sigma[x_i \leftarrow x_{i+1}]$.



Malheureusement, cette heuristique naïve ne conduit pas en général à une représentation compilée en $\text{EADT}_\wedge$ de « petite taille ». Elle revient à construire à chaque étape une clause affine de taille 2 en choisissant aléatoirement ses deux variables parmi les variables de la formule ECNF en entrée. De plus, elle peut défavoriser la décomposition puisqu'en considérant une clause affine $x_i \oplus x_{i+1}$, il se peut que nous construisions artificiellement des liens entre des composantes connexes séparées au départ.

Dans notre compilateur, l'heuristique utilisée pour choisir les clauses affines de la forme $x \oplus \delta$ à la ligne 5 est basée sur le concept d'activité des variables (VSIDS, *Variable State Independant Decaying Sum*) [MMZ+01b]. L'idée de cette heuristique est d'éliminer les variables qui apparaissent le plus dans les conflits lors de la recherche de solution. Ces variables étant généralement des variables clés du problème, les considérer le plus haut possible lors de la recherche permet en général de réduire la taille de la forme compilée. Plus précisément, pour chaque clause étendue ϵ de Σ, le score de ϵ est calculé comme la somme des scores de chaque clause affine qu'elle contient, où le score d'une clause affine est la somme des scores VSIDS de ses variables, $\text{VSIDS}(\epsilon) = \Sigma_{l_i \in \epsilon} \text{VSIDS}(var(l_i))$. Nous sélectionnons une clause étendue ϵ de Σ de score maximal, et les variables de ϵ sont triées par ordre décroissant selon leur score VSIDS. La variable x sélectionnée est la première variable de la liste ainsi créée. La clause affine δ que

nous créons est formée des $k-1$ variables suivantes de la liste. Notons que sélectionner uniquement les variables de la même clause étendue ϵ de Σ permet d'éviter de générer des connexions supplémentaires entre les variables qui ne sont pas déjà connectées dans le graphe primal de Σ. Ceci est donc fait dans le but de ne pas diminuer le nombre de composantes connexes qu'il est possible de trouver.

Nous nous efforçons aussi de réduire la formule ECNF courante à chaque étape de remplacement de x par δ ou par $\delta \oplus \top$: nous supprimons les littéraux redondants qui ont pu apparaître suite à l'ajout de la clause affine δ pour remplacer la variable x. Nous supprimons aussi des clauses étendues construites les clauses affines rendues contradictoires par le remplacement de x par δ (respectivement $\delta \oplus \top$).

Nous tirons enfin avantage d'une méthode simple de filtrage utilisée uniquement dans les premiers nœuds de l'arbre de recherche. Cette méthode consiste à calculer les clauses affines impliquées par la formule ECNF donnée initialement Σ. Une méthode pour détecter des clauses affines « simples » dans une ECNF est d'utiliser la propagation unitaire (voir [LM14]). L'idée est de rechercher des équivalences entre les littéraux de Σ. Pour cela, pour toutes les variables x de la formule Σ, nous retenons les variables propagées par $x$et nous faisons de même avec le littéral $\neg x$. Si un même littéral y fait partie des littéraux propagés par la décision x (respectivement $\neg x$) et que sa négation $\neg y$ apparaît dans la liste des littéraux propagés $\neg x$ (respectivement x), alors nous avons bien $x \Leftrightarrow y$ (respectivement $\neg x \Leftrightarrow y$). Nous en déduisons alors que la clause $\delta = x \oplus y \oplus \top$ (respectivement, $\delta x \oplus y$) est impliquée par Σ. Cette technique permet donc de détecter de façon peu coûteuse des clauses affines de taille deux (et certaines de taille trois).

Reconnaître une telle clause affine δ impliquée par la formule Σ permet de couper l'une des branches de l'arbre de recherche. En effet, comme Σ est cohérente et que $\Sigma \models \delta$, on sait que $\Sigma \wedge \neg\delta \models \bot$. Il est donc inutile d'explorer la branche $\delta \oplus \top$ de l'arbre. Plus cette clause affine est trouvée haut dans l'arbre, plus cela permet de réduit l'espace de recherche et, par conséquent, de réduire la taille de l'arbre final.

Exemple 67 *Illustrons maintenant le fonctionnement de l'algorithme* eadt *en le déroulant pas à pas sur la formule* CNF Σ *suivante :*

$$\begin{array}{ll} \neg x \vee y \vee \neg z & \wedge \\ x \vee y \vee z & \wedge \\ y \vee t \vee u & \end{array}$$

eadt(Σ) *engendre la formule* EADT$_\wedge$ *représentée par la figure 4.4.*

La formule Σ est bien cohérente et ne possède qu'une composante connexe. Nous construisons donc un nœud de décision affine $N_1 = \langle \delta_1, N_{1-}, N_{1+} \rangle$. Pour cela, l'algorithme a choisi la clause affine unitaire $\delta_1 = y$. Nous continuons vers le fils N_{1+}. Puisque la clause affine est unitaire, toute occurrence de la variable y est remplacée par $\top$ dans Σ. Comme y apparaît positivement dans les trois clauses de Σ, après simplification $\Sigma = \varnothing$ est donc la compilation termine et nous créons une feuille $\top$.

L'algorithme développe maintenant le nœud N_{1-}, dans lequel nous remplaçons toute occurrence de y par $\bot$. Après simplification, nous obtenons la formule CNF Σ_1 *suivante :*

$$\begin{array}{ll} \neg x \vee \neg z & \wedge \\ x \vee z & \wedge \\ t \vee u & \end{array}$$

Nous remarquons que Σ_1 possède deux composantes connexes et que la condition de décomposabilité affine est vérifiée. En effet, les variables t et u n'apparaissent dans aucune clause en commun avec les variables x et z. De plus, y est unitaire (et n'apparait pas dans la formule obtenue). Nous pouvons donc considérer deux formules ECNF *séparément : la formule $\Sigma_{11} = (\neg x \vee \neg z) \wedge (x \vee z)$ d'un côté et la formule $\Sigma_{12} = t \vee u$ de l'autre.*

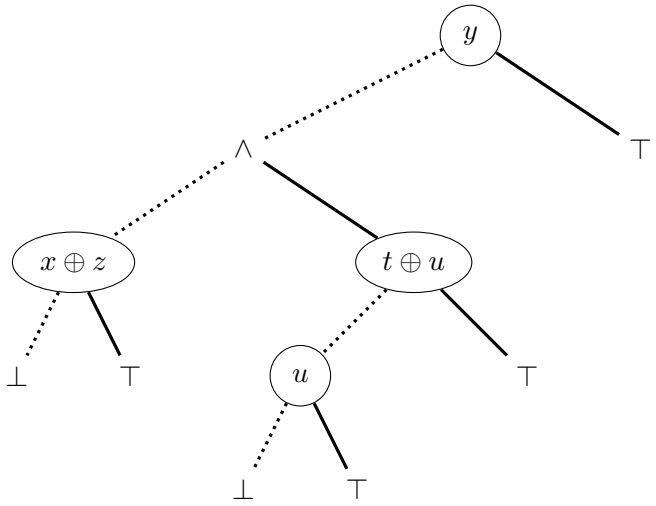


FIGURE 4.4 – Formule résultant de l'exécution de eadt sur Σ

Nous compilons maintenant la formule Σ_{12}. Σ_{12} est bien cohérente et ne possède qu'une composante connexe, nous créons donc un nouveau nœud de décision affine $N_2 = \langle \delta_2, N_{2-}, N_{2+} \rangle$. La clause affine $\delta_2 = t \oplus u$ est sélectionnée est la variable t est choisie pour être remplacée. Pour le fils N_{2+}, comme $\delta_2 = t \oplus u \equiv t \Leftrightarrow u \oplus \top$, les occurrences de t dans Σ_{12} sont remplacées par le littéral $\neg u$ (logiquement équivalent à $u \oplus \top$). Nous obtenons la clause tautologique $\neg u \vee u$. Donc, après simplification Σ_{12} est vide. Nous créons alors une feuille $\top$. Pour le cas du fils N_{2-}, toute occurrence de t dans Σ_{12} est remplacée par u. Nous obtenons la formule $\Sigma_3 = u$. Cette formule étant cohérente et ne possédant qu'une composante connexe, nous ajoutons un nouveau nœud de décision affine $N_3 = \langle \delta_3, N_{3-}, N_{3+} \rangle$, avec la clause affine $\delta_3 = u$. Dans le fils N_{3+}, u est remplacé par $\top$ et Σ_3 est donc tautologique. Une feuille $\top$ est donc créée. Pour le fils N_{3-},u est remplacé par $\bot$ et Σ_3 est donc incohérente. Par conséquent, une feuille $\bot$ est créée.

Enfin, nous développons la formule Σ_{11}. Σ_{11} est cohérente et ne possède qu'une composante connexe. L'algorithme crée donc un nouveau nœud de décision affine $N_4 = \langle \delta_4, N_{4-}, N_{4+} \rangle$. La clause affine $\delta_4 = x \oplus z$ est sélectionnée. Dans le fils N_{4+}, les occurrences de la variables x dans Σ_{11} sont remplacées par la clause affine $z \oplus \top$. Après simplification, Σ_{11} est vide et tautologique, une feuille $\top$ est donc créée. Finalement, dans le fils N_{4-}, toute occurrence de x est remplacée par z dans Σ_{11}. Après simplification, Σ_{11} est contradictoire, nous créons alors une feuille $\bot$.

La formule $\texttt{EADT}_\wedge$ ainsi construite est bien logiquement équivalent à la formule ECNF Σ.

Une fois la compilation terminée, la formule $\mathsf{eadt}(\Sigma)$ obtenue peut être utilisée pour compter directement le nombre de modèles de Σ. En effet, notre approche consistant à créer une clause affine $x \oplus \delta$ avec laquelle nous remplaçons toute occurrence de la variable x par δ (ou $\delta \oplus \top$) dans la formule Σ, la variable x peut être considérée comme variable pivot de la clause affine $x \oplus \delta$ puisqu'elle n'apparaîtra plus dans les clauses affine suivantes. Cela est vrai pour l'ensemble des clauses affine créées sur le chemin entre la racine et une feuille.

Ainsi, les formules affines engendrées par notre algorithme sont triangulées par construction. Il suffit donc de connaître la profondeur p du chemin entre la racine et la feuille, ainsi que le nombre de variables de la composante connexe courante afin d'en connaître le nombre de modèles.

4.3.3 Expérimentations

Dans cette section nous présentons le protocole utilisé pour nos expérimentations, puis nous discutons des résultats obtenus.

Dans nos expérimentations, nous avons décidé de comparer notre approche par compilation en EADT$_\wedge$ pour le comptage de modèles à plusieurs approches proposées dans la littérature à cet effet. Pour cela, le protocole empirique que nous avons suivi est très proche de celui appliqué dans [Sch96] (et d'autres articles) où le but est de déterminer si (et quand) une approche pour l'implication clausale basée sur la compilation peut se montrer plus satisfaisante qu'une approche directe pour l'implication clausale, sans aucune phase de compilation.

Nous avons considéré plusieurs jeux d'essai constitués de formules CNF issus de différents domaines de la SAT LIBrary [5]. Nous avons uniquement sélectionné les jeux d'essais ayant plus d'une solution.

Pour chaque instance CNF Σ, nous avons généré 1000 requêtes. Chaque requête est un terme γ de 3 littéraux. Les 3 variables choisies pour générer le terme γ sont sélectionnées au hasard parmi l'ensemble des variables de Σ, suivant une distribution uniforme. La polarité de chaque littéral est aussi choisie avec une probabilité $\frac{1}{2}$. L'objectif est de compter le nombre de modèles de la formule conditionnée $\Sigma \mid \gamma$ pour chaque requête γ.

Nos expérimentations ont été effectuées sur un Quad-core Intel XEON X5550 avec 32Go de mémoire vive. Nous avons considéré un temps limite de 3 heures pour la phase de compilation, et un temps limite de 3 heures par requête pour répondre à chacune des 1000 requêtes lors de la phase en ligne. Selon ce protocole, nous avons examiné trois approches différentes :

— une approche directe (non compilée) : nous nous sommes basés sur le compteur de modèles de l'état de l'art, Cachet [6] [SBB+04b]. Ici, $\#F_{\text{Cachet}}$ est le nombre d'éléments de F_{Cachet}, l'ensemble des requêtes « réalisables », c'est-à-dire les requêtes de notre échantillon pour lesquelles Cachet a été capable de terminer avant la fin du temps limite (ou d'une erreur de segmentation). $\overline{Q}_{\text{Cachet}}$ indique le temps moyen nécessaire à Cachet pour calculer les requêtes réalisables, c'est-à-dire,

$$\overline{Q}_{\text{Cachet}} = \frac{1}{\#F_{\text{Cachet}}} \sum_{\gamma \in F_{\text{Cachet}}} Q_{\text{Cachet}}(\Sigma|\gamma)$$

où $Q_{\text{Cachet}}(\Sigma|\gamma)$ est le temps nécessaire à Cachet pour calculer $\|\Sigma|\gamma\|$.

— deux approches par compilation, une ciblant d-DNNF et l'autre EADT$_\wedge$, ont été utilisées. Nous avons en recours au compilateur c2d [7] afin de compiler les instances Σ en d-DNNF [8] et notre propre compilateur eadt pour compiler en EADT$_\wedge$. Pour chaque langage $\mathcal{L}$ parmi d-DNNF et EADT$_\wedge$, Σ a tout d'abord été traduite en sa forme compilée $\Sigma^* \in \mathcal{L}$. Le temps de compilation $C_\mathcal{L}$ nécessaire pour calculer Σ^* et le temps de réponse moyen aux requêtes $\overline{Q}_\mathcal{L}$ ont été mesurés. Pour chaque approche, nous avons aussi calculé deux ratios :

$$\alpha_\mathcal{L} = \frac{\overline{Q}_\mathcal{L}}{\overline{Q}_{\text{Cachet}}} \text{ et } \beta_\mathcal{L} = \left\lceil \frac{C_\mathcal{L}}{\overline{Q}_{\text{Cachet}} - \overline{Q}_\mathcal{L}} \right\rceil$$

Intuitivement, $\alpha_\mathcal{L}$ indique le temps en ligne gagné grâce à la compilation : plus il est petit, mieux c'est. $\beta_\mathcal{L}$ capture le nombre de requêtes nécessaire pour amortir le temps de compilation. L'approche $\mathcal{L}$ est utile seulement si $\alpha_\mathcal{L} < 1$. Par convention, $\beta_\mathcal{L} = +\infty$ quand $\alpha_\mathcal{L} \geq 1$.

5. SATLIB est disponible sur le site www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html
6. Cachet est disponible à l'adresse suivante : www.cs.rochester.edu/~kautz/Cachet/index.htm
7. c2d est disponible à reasoning.cs.ucla.edu/c2d/
8. Nous avions aussi prévu d'utiliser le compilateur d-DNNF Dsharp [MMBH12] mais malheureusement, nous avons rencontré le même problème celui que mentionné dans [Vor13] pour le lancer, ce qui nous a empêchés de l'utiliser.

Quand nous avons utilisé notre compilateur eadt, nous avons borné la taille des clauses affines générées à $k = 2$ et utilisé la méthode de filtrage pour la reconnaissance des clause affines impliquées jusqu'à une profondeur de 5. Dans ce cas, le remplacement de variable est particulièrement simple (on remplace une variable par une variable et pas par une clause affine plus complexe). Les formules obtenues sont donc des formules CNF quand la formule de départ est elle-même une formule CNF. Ceci permet de tirer avantage des techniques implémentées dans le solveur intégré au compilateur pour détecter les formules contradictoires rencontrées. Bien que nous ayons modifié ce solveur pour qu'il traite des instances ECNF, il est plus efficace en pratique quand il opère sur des formules CNF.

Le tableau 4.3 présente les résultats obtenus. Chaque ligne correspond à une instance CNF Σ identifiée par la colonne la plus à gauche. Les deux premières colonnes donnent respectivement le nombre $\#var$ de variables de Σ et le nombre $\#cla$ de clauses de Σ, les autres colonnes indiquent les valeurs mesurées. Les différents temps de calculs rapportés sont exprimés en secondes.

Nous pouvons observer que chacune des approches compilées se montre utile lorsque la phase de compilation aboutit. D'une part, pour chacune des approches par compilation dans $\mathcal{L}$, les 1000 requêtes de comptage ont été « réalisables » quand la compilation s'est terminée dans les temps. Pour cette raison, nous n'avons pas donné dans le tableau le nombre $\#F_{\mathcal{L}}$ de requêtes réalisables. En revanche, le nombre de requêtes réalisables par l'approche directe est parfois significativement plus petit que 1000, et l'écart-type du temps de réponse aux requêtes pour de telles requêtes est souvent bien plus grand que l'écart correspondant pour les approches compilées. D'autre part, le nombre de requêtes β à considérer afin d'amortir le temps de compilation est fini pour toutes les instances considérées, sauf deux (une pour la compilation en d−DNNF et une pour la compilation en $\text{EADT}_{\wedge}$).

Les expérimentations ont aussi révélé que quelques instances de taille conséquente sont compilables. Quand la compilation aboutit, β est généralement petit. En conséquence, le temps de comptage en ligne peut diminuer de plusieurs ordres de grandeur. En particulier, la valeur optimale 1 pour le seuil de rentabilité β a été atteinte pour plusieurs instances quand le langage cible considéré est $\text{EADT}_{\wedge}$. Cela signifie que dans plusieurs cas, le temps passé à compiler la formule en $\text{EADT}_{\wedge}$ est rentabilisé dès la première requête de comptage. Ainsi, le compilateur eadt se montre aussi compétitif en tant que compteur exact de modèles.

Finalement, nos expérimentations montrent que la compilation en $\text{EADT}_{\wedge}$ donne de meilleurs résultats que la compilation en d−DNNF dans de nombreux (mais pas tous les) cas. Quand la compilation réussit dans les deux cas, les nombres de nœuds des formules $\text{EADT}_{\wedge}$ et d−DNNF sont du même ordre de grandeur, mais la formule $\text{EADT}_{\wedge}$ est légèrement plus efficace pour traiter les requêtes de comptage.

4.4 Conclusion

Nous avons introduit des nouveaux langages cibles de compilation s'appuyant sur la notion de nœud de décision affine. Nous avons montré que le langage $\text{EADT}_{\wedge}$ est un langage de compilation très attractif quand le comptage de modèles est une requête clé. En particulier, le langage $\text{EADT}_{\wedge}$ offre les mêmes requêtes que d−DNNF. Le sous-ensemble ADT de $\text{EADT}_{\wedge}$ satisfait toutes les requêtes et mêmes transformations que celles offertes par le langage $\text{OBDD}_{<}$. De plus, $\text{OBDD}_{<}$ n'est pas au moins aussi succinct que ADT, ce qui place ADT comme un challenger possible du langage $\text{OBDD}_{<}$. En pratique, l'approche par compilation en $\text{EADT}_{\wedge}$ pour compter les modèles apparaît compétitive quand on la compare au compteur Cachet de modèles et à l'approche par compilation en d−DNNF. En effet, empiriquement, lorsque la compilation aboutit, toutes les requêtes de comptage ont pu ensuite être réalisées dans le temps imparti. Nous avons montré que, dans plusieurs cas, la compilation vers le langage $\text{EADT}_{\wedge}$ suivie de la requête de comptage de modèles est même plus rapide qu'une approche directe de comptage de modèles. Cela signifie que dans certains cas, le temps de compilation est compensé dès la première requête.

Instance			Cachet		c2d				eadt			
name	$\#var$	$\#cla$	$\#F$	$\overline{Q}$	C	$\overline{Q}$	α	β	C	$\overline{Q}$	α	β
All interval series												
ais6	61	581	1000	0.531	1.23	4.10^{-5}	8.10^{-7}	2	0.01	$< 1.10^{-7}$	$< 2.10^{-7}$	**1**
ais8	113	1520	866	0.540	3.04	2.10^{-4}	3.10^{-4}	5	0.24	1.10^{-5}	2.10^{-5}	**1**
ais10	181	3151	325	0.578	12.3	1.10^{-3}	2.10^{-3}	21	7.69	1.10^{-4}	2.10^{-4}	**13**
ais12	265	5666	80	0.573	-	-	-	-	410	1.10^{-3}	2.10^{-3}	**717**
Bounded model checking												
bmc-ibm-2	2810	11683	1000	0.569	-	-	-	-	0.37	2.10^{-5}	4.10^{-5}	**1**
bmc-ibm-3	14930	72106	999	13.04	412	0.93	7.10^{-2}	34	180	6.10^{-3}	5.10^{-4}	**13**
bmc-ibm-4	28161	139716	1000	5.412	1128	9.09	1.679	$+\infty$	-	-	-	-
Planning												
bw_large.a	459	4675	1000	0.537	15.05	2.10^{-5}	4.10^{-5}	28	$< 1.10^{-4}$	$< 1.10^{-7}$	$< 2.10^{-7}$	**1**
bw_large.b	1087	13772	1000	0.612	48.88	5.10^{-5}	8.10^{-5}	79	0.01	$< 1.10^{-7}$	$< 2.10^{-7}$	**1**
bw_large.c	3016	50457	996	2.452	283.3	1.10^{-4}	6.10^{-5}	115	0.16	1.10^{-5}	4.10^{-6}	**1**
bw_large.d	6325	131973	896	31.44	-	-	-	-	1.9	2.10^{-5}	6.10^{-7}	**1**
(bw) medium	116	953	1000	0.526	2.39	2.10^{-5}	4.10^{-5}	4	$< 1.10^{-4}$	$< 1.10^{-7}$	$< 2.10^{-7}$	**1**
(bw) huge	459	7054	1000	0.543	15.11	2.10^{-5}	4.10^{-5}	27	$< 1.10^{-4}$	$< 1.10^{-7}$	$< 2.10^{-7}$	**1**
Hanoi												
hanoi4	718	4934	505	0.557	559.6	3.10^{-5}	5.10^{-5}	1004	0.13	$< 1.10^{-7}$	$< 2.10^{-7}$	**1**
hanoi5	1931	14468	440	0.619	2240	8.10^{-5}	1.10^{-4}	3621	1.1	1.10^{-5}	2.10^{-5}	**1**
logistics.a	828	6718	993	1.266	-	-	-	-	6757	2.12	1.676	$+\infty$
Circuit												
ssa7552-038	1501	3575	1000	0.634	20.99	0.042	0.065	**35**	-	-	-	-

TABLE 4.3 – Quelques résultats expérimentaux

Chapitre 5

La compilation de réseaux de contraintes en graphes de décision décomposables multivalués

Sommaire

5.1 Le langage MDDG 101
5.2 MDDG et ses voisins 105
5.3 Un compilateur CN-to-MDDG de type descendant 107
5.3.1 Les contraintes universelles 108
5.3.2 Le *caching* 109
5.3.3 L'heuristique de choix de variable 110
5.3.4 L'algorithme de compilation 111
5.4 Expérimentations 113
5.4.1 Le cadre expérimental 113
5.4.2 Les résultats obtenus 114
5.5 Conclusion 116

Dans ce chapitre, une nouvelle approche de compilation de réseaux de contraintes est présentée. Un algorithme descendant de compilation de réseaux de contraintes, à domaines finis, est décrit et évalué. Ce compilateur cible un nouveau langage de compilation pour les réseaux de contraintes : le langage des graphes de décision décomposables multivalués (MDDG). Ce langage peut être vu comme une généralisation du langage cible Decision-DNNF au cas où les variables figurant dans les contraintes ne sont pas booléennes. Nous listons les différentes propriétés de la carte de compilation qui sont offertes par ce langage. Nous montrons que notre approche permet le comptage de modèles de réseaux de contraintes de manière relativement efficace. Ce travail a donné lieu à une publication dans les actes de la conférence IJCAI 2015 [KLMT15].

Le formalisme des réseaux de contraintes permet souvent de modéliser plus simplement les problèmes que la logique propositionnelle, de par la diversité possible des contraintes que l'on peut utiliser et par le fait que les variables d'un réseau de contraintes ne sont pas forcément booléennes. Comme en logique propositionnelle, le comptage de modèles est une requête clé pour la résolution de divers problèmes représentés par des réseaux de contraintes. Toutefois, à l'instar du comptage en logique propositionnelle (qui peut être vu comme un cas particulier du comptage des modèles d'un réseau de contraintes), cette requête est coûteuse dans le cas général.

Il est donc intéressant de tenter de définir une méthode moins coûteuse que celles qui existent pour compter les modèles d'un réseau de contraintes. Il est à noter que peu de méthodes permettent jusqu'ici de répondre à la requête de comptage. Ainsi, l'algorithme de comptage le plus répandu consiste à énumérer l'ensemble des modèles du réseau et d'en compter le nombre. Cette approche est notablement inefficace (et clairement impraticable dès que le nombre de modèles est élevé).

Comme en logique propositionnelle, on peut réaliser le comptage des modèles d'un réseau de contraintes en suivant une approche par compilation. La compilation de réseaux de contraintes est un sujet qui a déjà été considéré mais qui reste peu développé en comparaison à la compilation de formules propositionnelles. Les travaux réalisés sont souvent plus orientés vers des aspects théoriques (définition de langages de compilation, étude de leurs propriétés) que pratiques (conception et évaluation de compilateurs). En particulier, il y a peu de compilateurs fonctionnels et maintenus permettant de compiler un réseau de contraintes vers un langage offrant le comptage de modèles en temps polynomial.

5.1 Le langage MDDG

Dans cette section, nous présentons le langage des graphes de décision décomposables multivalués (MDDG) qui est le langage cible de notre compilateur de réseau de contraintes. Dans un premier temps, dans le but de fixer un certain nombre de notions utilisées dans la suite, nous décrivons le langage (plus général) des graphes de décision multivalués (MDG).

Définition 66 (MDG) *Soit un ensemble fini $\mathcal{X}$ de variables à domaines finis. Le langage MDG des graphes de décision multivalués (à nœuds de décision à lecture unique) sur $\mathcal{X}$ est l'ensemble des graphes Δ orientés sans circuit à racine unique, tels que chaque feuille est étiquetée par $\top$ (vrai) ou $\bot$ (faux), et chaque nœud interne N est soit un nœud de conjonction $N = \wedge(N_1, \ldots, N_i)$ où les N_i représentent les nœuds fils de N, soit un nœud de décision $N = \langle X_i \mid x_{i1}, \ldots, x_{ij}\rangle$ associé à une variable $X_i \in \mathcal{X}$, de domaine $D_i = \{x_{i1}, \ldots, x_{ij}\}$. Tout chemin de la racine de Δ a une feuille ne peut comporter qu'au plus un arc étiqueté par une affectation élémentaire portant sur X_i, pour toute variable $X_i \in \mathcal{X}$.*

Chaque nœud de décision $N = \langle X_i \mid x_{i1}, \ldots, x_{ij}\rangle$ d'un graphe MDG peut être vu comme un nœud $\vee$ déterministe $N = \vee(N_1, \ldots, N_j)$ tel que l'arc entre N et N_k ($k \in \{1, \ldots, j\}$) est étiqueté par l'affectation élémentaire $\{\langle X_i, x_{ik}\rangle\}$.

Exemple 68 *Nœud de décision Soit le nœud de décision $N = \langle X_4 \mid 0, 2, 4\rangle$ portant sur la variable X_4 ayant pour domaine $D_4 = \{0, 2, 4\}$. Le nœud de décision N et sa représentation en nœud $\vee$ déterministe sont schématisés sur les figures 5.1a et 5.1b.*

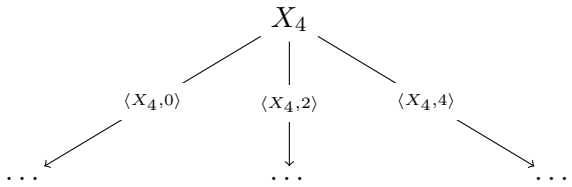
Pour chaque nœud N d'un graphe MDG Δ, l'ensemble des variables de N, noté $Var(N)$, est défini inductivement comme suit :

— si N est une feuille, alors $Var(N) = \varnothing$;
— si $N = \wedge(N_1, \ldots, N_i)$ est un nœud $\wedge$, alors $Var(N) = \bigcup_{k=1}^{i} Var(N_k)$;
— si N est un nœud de décision $N = \vee(N_1, \ldots, N_j)$ associé à une variable X_i, alors $Var(N) = \{X_i\} \cup \bigcup_{k=1}^{j} Var(N_k)$.

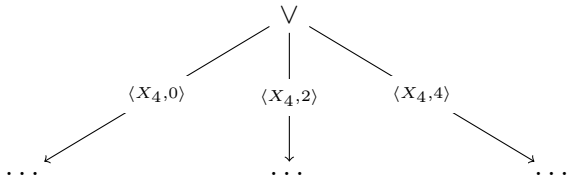
L'évaluation d'une formule MDG se fait en un seul parcours du graphe, de la manière suivante :

Définition 67 (Sémantique d'un graphe MDG) *Soit $\mathbf{s}$ une instantiation complète sur l'ensemble de variables $\mathcal{X}$ et soit Δ une formule MDG définie sur $\mathcal{X}$, ayant pour racine N. Nous définissons la valeur de vérité du graphe Δ pour l'instantiation $\mathbf{s}$, noté $eval(N, \mathbf{s})$, de manière inductive comme suit :*

— *si N est une feuille $\top$, alors $eval(N, \mathbf{s}) = \top$;*



(a) Représentation du nœud de décision



(b) Représentation en nœud $\vee$ déterministe

— *si N est une feuille $\bot$, alors eval$(N, \mathbf{s}) = \bot$;*
— *si $N = \wedge(N_1, \ldots, N_i)$ est un nœud $\wedge$, alors eval$(N, \mathbf{s}) = \wedge($eval$(N_1, \mathbf{s}), \ldots,$ eval$(N_i, \mathbf{s}))$;*
— *si N est un nœud de décision $N = \vee(N_1, \ldots, N_j)$ associé à la variable X, alors eval$(N, \mathbf{s}) =$ eval$(N_k, \mathbf{s})$ avec $\langle X, x_k\rangle \in \mathbf{s}$.*

Soit $\mathbf{s}$ une instanciation complète sur $\mathcal{X}$ et Δ un graphe MDG défini sur $\mathcal{X}$ et de racine N. L'instantiation $\mathbf{s}$ est une solution (ou modèle) de Δ si et seulement si le graphe Δ *eval*$(N, \mathbf{s})$ est évalué à vrai.

Il est aisé de montrer que MDG ne permet pas le comptage de modèles en temps polynomial sauf si P = NP. Il faut donc imposer des conditions supplémentaires pour cela. Nous nous intéressons dans la suite à un sous-ensemble particulier de MDG, le langage des graphes de décision décomposables multivalués (MDDG). Le langage MDDG est le sous-ensemble MDG contenant des graphes où les nœuds de conjonction sont décomposables.

Définition 68 (MDDG) *Un graphe MDDG Δ est un graphe MDG dans lequel tout nœud de conjonction $N = \wedge(N_1, \ldots, N_i)$ est décomposable : pour tous $k, l \in \{1, \ldots, i\}$, tels que $k \neq l$, nous avons Var$(N_k) \cap$ Var$(N_l) = \varnothing$.*

Exemple 69 *Dans la suite de ce chapitre, nous considérons à titre d'exemple le réseau de contraintes $\mathcal{N}$ défini sur quatre variables X_1, X_2, X_3 et X_4, chacune étant associée au même domaine $D_1 = D_2 = D_3 = D_4 = \{0, 1, 2\}$. $\mathcal{N}$ porte sur trois contraintes C_1, C_2 et C_3, caractérisées par les expressions suivantes :*
— *$C_1 = (X_1 \neq X_2)$;*
— *$C_2 = (X_2 = 0) \vee (X_2 = 1) \vee (X_2 = X_3 + X_4 + 1)$;*
— *$C_3 = (X_3 > X_4)$.*
La figure 5.2 représente le graphe d'incidence du réseau de contrainte $\mathcal{N}$.

Par exemple, le graphe MDDG donné à la figure 5.3 est équivalent au réseau de contraintes $\mathcal{N}$ donné dans l'exemple 69 : les deux représentations ont le même ensemble de solutions. Notons que pour des raisons de lisibilité, les arcs conduisant à une feuille $\bot$ ne sont pas représentés sur la figure et la feuille $\top$ est dupliquée. De plus, les nœuds avec un seul fils sont supprimés et leurs étiquettes (dans le cas des nœuds de décision) sont regroupées avec celles de leur arc incident. En effet, durant la phase de compilation, il arrive que des affectations élémentaires soient obtenues par propagation. Dans ce cas, les nœuds de décision ne sont pas explicitement construits. En effet, comme la valeur est obtenue par

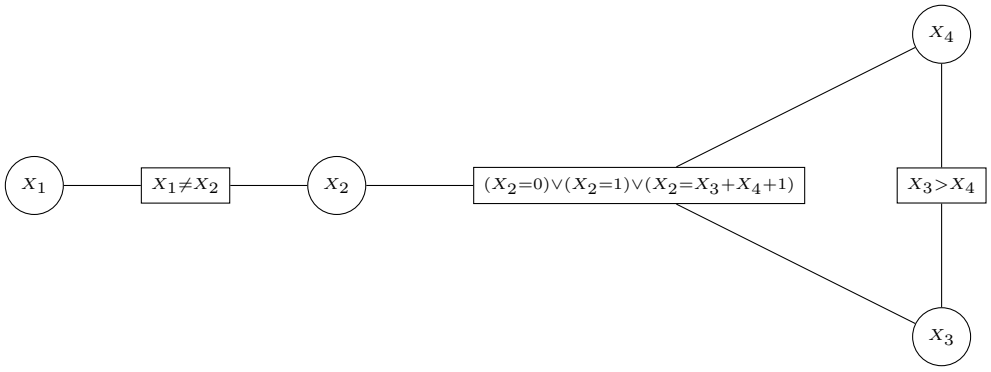


FIGURE 5.2 – Graphe d'incidence du réseau de contraintes $\mathcal{N}$

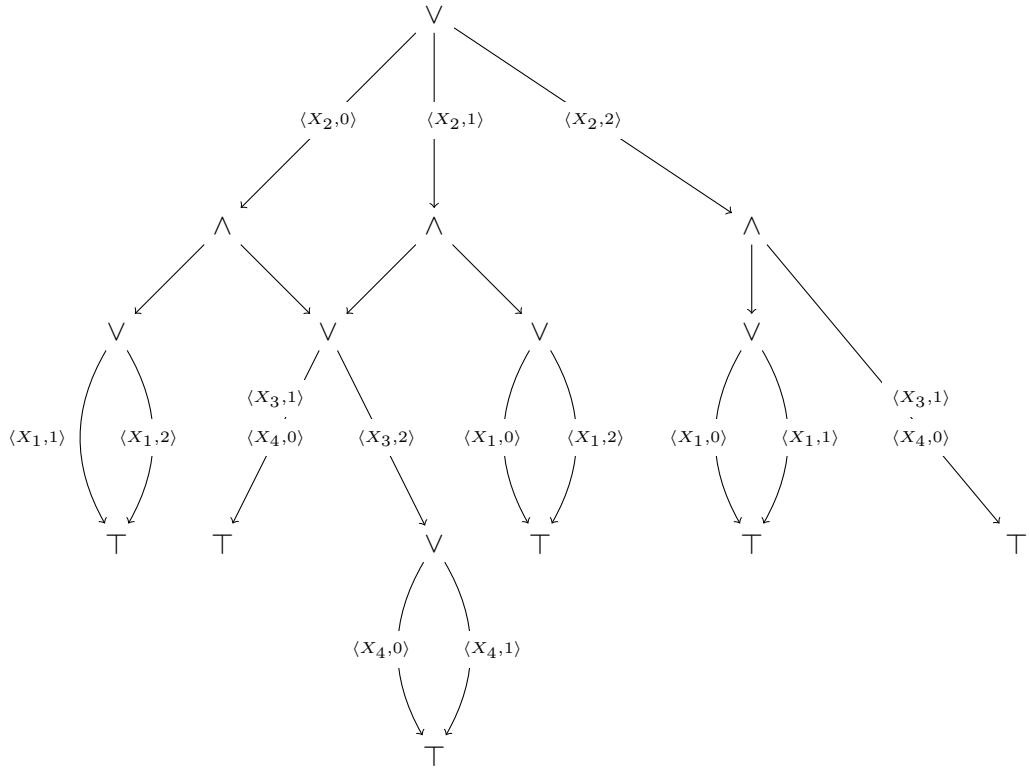


FIGURE 5.3 – Graphe MDDG équivalent au réseau de contraintes de l'exemple 69.

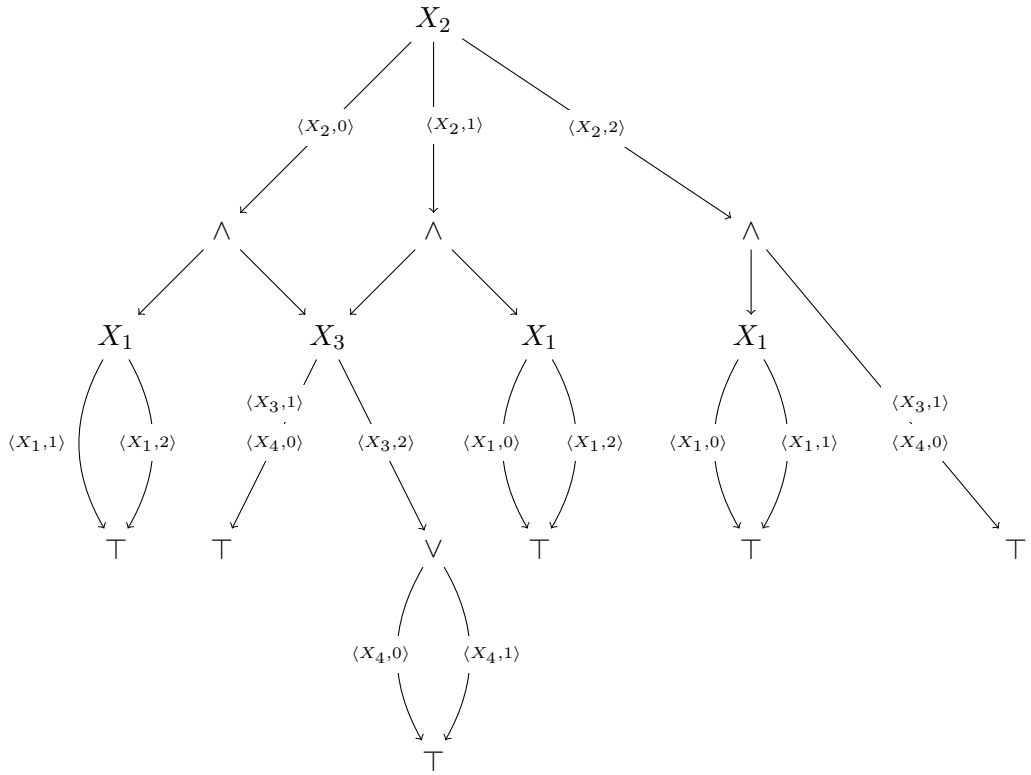


FIGURE 5.4 – Un MDDG équivalent au CN de l'exemple 69.

propagation, cela signifie que la variable ne pourra pas avoir d'autre valeur que celle propagée. Il est donc inutile de créer un nœud décision portant sur cette variable puisqu'il n'aurait qu'un seul fils.

Le nœud racine du graphe MDDG obtenu est le premier nœud de décision $N_1 = \langle X_2 \mid 0, 1, 2 \rangle$. Il est étiqueté par la variable X_2. Nous remarquons qu'une fois la variable X_2 affectée, la formule est décomposable en deux composantes connexes distinctes. D'un côté, nous avons le réseau de contraintes portant sur l'ensemble de variables $\{X_1\}$ et de l'autre celui portant sur l'ensemble de variables $\{X_3, X_4\}$.

5.2 MDDG et ses voisins

Le langage MDDG possède des similarités avec le langage MDD considéré dans [AFNP14], ainsi qu'avec le langage AOMDD considéré dans [MD06, MDM08], mais il ne coïncide avec aucun d'entre eux.

Ainsi, les langages MDD et MDDG ne sont pas comparables en terme d'inclusion. D'un côté, une représentation d'un réseau de contraintes en MDD est une structure non déterministe. En effet, dans le langage MDD plus d'un arc sortant d'un nœud de décision étiqueté par une variable X_i peut être étiqueté par une même valeur du domaine D_i de X_i, alors que les nœuds de décision des MDDG sont toujours déterministes. Chaque nœud de décision, d'une formule MDDG, portant sur une variable X_i est l'origine de $|D_i|$ arcs. D'un autre côté, les représentations en MDDG autorisent des nœuds de conjonction décomposables, tandis que les nœuds internes des MDD sont tous des nœuds de décision.

De la même manière, les langages AOMDD et MDDG ne sont pas comparables du point de vue de l'inclusion. Le langage AOMDD est adapté à la compilation de modèles graphiques, en particulier des réseaux bayésiens. Il permet la représentation de fonctions non booléennes, comme des fonctions d'utilité ou de coût, des distributions de probabilités discrètes, etc. Le langage MDDG ne le permet pas (la sémantique d'un graphe MDDG est booléenne). De plus, les AOMDD sont des structures ordonnées qui respectent un pseudo-arbre induit par un ordre prédéfini d'élimination des variables. C'est un pré-requis indispensable pour assurer la canonicité des AOMDD. À l'opposé, les MDDG n'imposent pas une telle condition. Les représentations en MDDG ne sont pas canoniques. Plusieurs formules MDDG différentes peuvent représenter un même réseau de contraintes.

Nous notons que le langage booléen Decision-DNNF [OD14, FM06] est un sous-ensemble du langage des MDDG où chaque variable possède un domaine booléen. En plus du gain d'expressivité obtenu en autorisant des domaines non booléens, le langage MDDG offre essentiellement les « mêmes requêtes et les mêmes transformations » que le langage Decision-DNNF (évidemment, une fois étendue au cas des réseaux de contraintes, de façon à prendre en compte le fait que les variables ne sont pas forcément booléennes). On trouve parmi elles le comptage de solutions (éventuellement pondérées), l'énumération de solutions et le conditionnement. Les algorithmes en temps polynomial utilisés pour réaliser ces requêtes et transformations quand le langage Decision-DNNF est ciblé peuvent être étendus facilement au langage MDDG.

La carte de compilation des langages booléens peut ainsi être étendue aux différents langages de compilation pour les réseaux de contraintes. Les deux tableaux indiquent respectivement les requêtes 5.5 et les transformations 5.6 offertes par les langages propositionnels qui ont fait l'objet de notre attention dans cette thèse. Nous les complétons en introduisant les langages MDD et MDDG.

Pour montrer que **CT** est satisfait, il suffit de mettre en évidence un algorithme de comptage de modèles qui opère en temps polynomial sur le réseau de contraintes donné en entrée. Le compteur de modèles sharpmddg donné à l'algorithme 7 montre comment compter les modèles d'un graphe MDDG. Il s'agit d'un algorithme récursif ascendant, qui opère des feuilles du graphe vers sa racine. Si le nœud courant est un nœud $\bot$, alors l'algorithme retourne 0 modèles. Si le nœud est un nœud $\top$, alors le nombre de modèles de ce nœud est 1. Si N est une nœud de conjonction décomposable, alors le nombre de

$\mathcal{L}$	**CO**	**VA**	**CE**	**IM**	**EQ**	**SE**	**CT**	**ME**
EADT$_\wedge$	$\surd$	$\surd$	$\surd$	$\surd$	?	?	$\surd$	$\surd$
EDT	$\surd$	$\surd$	$\surd$	$\surd$	?	?	$\surd$	$\surd$
ADT	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
DT	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
OBDD$_<$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
d-DNNF	$\surd$	$\surd$	$\surd$	$\surd$	?	$\circ$	$\surd$	$\surd$
MDD	$\circ$	$\circ$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$
MDDG	$\surd$	$\surd$	$\surd$	$\surd$	?	$\circ$	$\surd$	$\surd$

FIGURE 5.5 – Requêtes. $\surd$ signifie “satisfait” et $\circ$ signifie “ne satisfait pas sauf si P = NP.”

$\mathcal{L}$	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
EADT$_\wedge$	$\surd$	$\circ$	?	$\circ$	$\circ$	$\circ$	?	?
EDT	$\surd$	$\circ$	?	$\circ$	$\circ$	$\circ$	?	?
ADT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
DT	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
OBDD$_<$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\circ$	$\surd$	$\surd$
d-DNNF	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?
MDD	$\surd$	$\circ$	$\circ$	$\surd$	$\surd$	$\surd$	$\surd$	?
MDDG	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	?

FIGURE 5.6 – Transformations. $\surd$ signifie “satisfait” et $\circ$ signifie “ne satisfait pas sauf si P = NP.”

Algorithme 7 : sharpmddg(Σ_{mddg})

entrée : La racine $N = R_{\Sigma_{mddg}}$ d'un graphe MDDG
sortie : Le nombre de modèles de Σ_{mddg}

```
1  si N = leaf(⊤) alors
2      return 1
3  si N = leaf(⊥) alors return 0;
4  si N = ⋀_{i=1}^{k} N_i alors
5      #models = 1;
6      pour tous les N_i de 1 à k faire
7          #models = #models × sharpmddg(N_i);
8      return #models;
9  sinon
10     //N = ⟨X_i | N_i1, ..., N_ij⟩ est alors un nœud de décision;
11     #models = 0;
12     pour tous les N_ik de i1 à ij faire
13         cadrage = 1;
14         pour chaque X_j variable de Var(N) \ Var(N_ik) faire
15             cadrage = cadrage× | D_j |;
16         #models = #models + sharpmddg(N_ik) × cadrage;
17     return #models;
```

modèles de ce nœud correspond au produit des nombres de modèles de ses fils. Enfin, si N est un nœud de décision, alors le nombre de modèles de N est représenté par la somme des nombres de modèles de ses fils, chacun multiplié par un facteur de cadrage qui lui est propre et est égal au produit des cardinaux des domaines des variables apparaissant dans N mais pas dans le fils considéré (autrement dit, apparaissant dans un des frères du fils considéré, mais pas dans le fils lui-même).

5.3 Un compilateur CN-to-MDDG de type descendant

Dans cette section, nous présentons d'abord les ingrédients utilisés dans notre compilateur cn2mddg qui associe un réseau de contraintes donné en entrée à un graphe MDDG ayant les mêmes solutions. Nous présentons ensuite le compilateur lui-même.

Le compilateur cn2mddg que nous avons développé est de type descendant. Il prend en entrée un réseau de contraintes représenté sous le format XCSP 2.1 [RL09], et retourne une représentation MDDG équivalente à celui-ci. Notre compilateur prend en compte toutes les caractéristiques basiques offertes par le format XCSP 2.1. En particulier, les contraintes peuvent être représentées en intension (en tant que « prédicats ») ou en extension (en tant que « relations »). Cependant, seules trois contraintes globales seulement sont supportées par notre compilateur dans sa forme actuelle : *allDifferent*, *weightedSum* (c'est-à-dire les contraintes linéaires), et *element*. Il est évidemment aussi possible d'encoder toutes ces contraintes comme des « relations », mais dans ce cas il n'est pas possible de profiter des propagateurs dédiés à ces contraintes globales.

L'architecture de notre compilateur est similaire à celle des compilateurs de type descendant pour les domaines booléens, comme c2d (http://reasoning.cs.ucla.edu/c2d/), ou Dsharp

(`http://www.haz.ca/research/dsharp/`), tous deux ayant pour cible le langage `Decision-DNNF`.

Notre compilateur est basé sur un algorithme de recherche de solutions, en suivant la trace d'un solveur [HD07]. Celui-ci n'est toutefois pas un solveur standard, mais en fait un solveur adapté afin d'améliorer les performances du processus de compilation.

Ce solveur utilise des ingrédients similaires à ceux proposés dans `c2d` et dans `Dsharp`, dont l'analyse de conflits pour guider la recherche, la propagation de contraintes afin de simplifier la formule, la mise en cache (*caching*) de composantes afin d'éviter la duplication de parties identiques pour la forme compilée. Enfin, nous utilisons une heuristique de choix de variable dynamique, comme `Dsharp`, qui tire parti de l'heuristique de choix de variable *vsads*). A contrario, dans `c2d` l'heuristique de choix de variable est statique.

La méthode de *caching* et l'heuristique de choix de variable utilisées dans cn2mddg sont spécifiques à la nature de l'entrée (un réseau de contraintes `CN`), qui est effectivement plus structurée qu'une formule propositionnelle exprimée dans un format dit « plat », comme `CNF`. De plus, cn2mddg exploite une méthode spécifique pour gérer les contraintes universelles. La gestion des contraintes universelles permet d'effectuer plus de simplifications lors de la compilation.

Nous décrivons de manière plus précise dans les sous-sections suivantes la façon dont nous pouvons utiliser ces divers ingrédients pour la compilation de réseaux de contraintes.

5.3.1 Les contraintes universelles

Les contraintes universelles sont des contraintes qui sont nécessairement satisfaites, quelles que soient les affectations des variables apparaissant dans leur portée, et cela suivant les domaines courants des variables. Par exemple, avec le réseau de contraintes donné dans l'exemple 5.2, quand la contrainte C_2 est conditionnée par une affectation élémentaire $\{\langle X_2, 0\rangle\}$ ou $\{\langle X_2, 1\rangle\}$, C_2 devient universelle.

À chaque étape de la compilation, les contraintes universelles sont détectées. La détection d'une contrainte universelle étant coûteuse en temps, il est contraignant d'effectuer ce test pour chaque contrainte à chaque nœud du graphe de compilation. L'universalité de chaque contrainte $C_j = (S_j, R_j) \in \mathcal{C}$ est donc testée uniquement s'il existe au moins un $X_i \in s_j$ tel que D_i a été réduit par propagation à la suite de la dernière instantiation élémentaire. Nous cherchons ainsi une instantiation $\mathbf{s}$ des variables dans la portée de C_j parmi les valeurs de leur domaine courant telle que $\mathbf{s}$ ne satisfait pas R_j. C_j est valide si et seulement si nous ne pouvons pas trouver une telle instantiation $\mathbf{s}$. Pour des raisons d'efficacité, $\mathbf{s}$ est cherchée d'une façon paresseuse. Une fois une instatciation $\mathbf{s}$ trouvée, elle est sauvegardée, puis reconsidérée en priorité lorsque l'universalité de C_j est à nouveau testée.

Une fois détectées, les contraintes universelles C_j sont simplement supprimées du réseau courant. La suite des traitements est donc simplifié puisque ces contraintes n'ont plus besoin d'être prises en compte. La suppression d'une contrainte dans un `CN` entraîne la suppression d'arêtes dans le graphe d'incidence du `CN`, cela favorise donc aussi la décomposition. Enfin, la suppression des contraintes universelles du problème influence l'heuristique de choix de variable puisque celle-ci est basée sur le graphe d'incidence du réseau de contraintes. Notons que dans les compilateurs ciblant le langage `Decision-DNNF`, la gestion des contraintes universelles revient simplement à ignorer toutes les clauses partageant un littéral avec l'interprétation partielle courante.

Exemple 70 *Soit $C = X_1 + X_2 \leq X_3$ une contrainte C portant sur les trois variables X_1, X_2 et X_3 ayant respectivement pour domaines $D_1 = D_2 = \{0, 1, 2\}$ et $D_3 = \{4, 5, 6\}$. Cette contrainte est représentée par la figure 5.7. Lorsque nous considérons l'assignation $X_2 = 1$, toute assignation des variables X_1 et X_3 satisfait la contrainte C. Donc lorsque X_2 est assignée à la valeur 1, la contrainte devient universelle, comme illustré à la figure 5.8.*

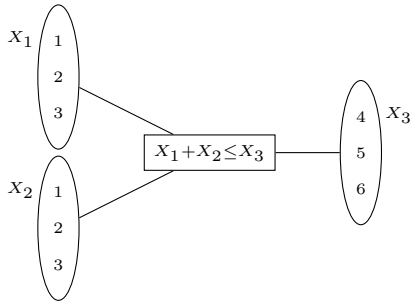


FIGURE 5.7 – Graphe de la contrainte C

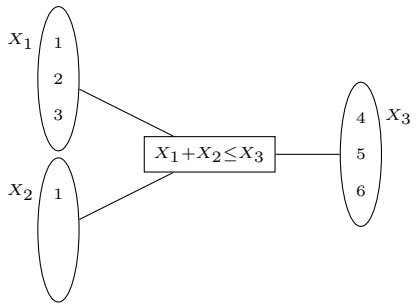


FIGURE 5.8 – Contrainte C rendue universelle après assignation de X_2 à 1

5.3.2 Le *caching*

La mise en cache (*caching*) est une technique cruciale pour les compilateurs calculant des représentations à base de graphes orientés sans circuit (DAG). Le but est d'éviter d'explorer le même sous-problème deux fois ou plus, et de dupliquer la partie compilée. En effet, du fait de l'interchangeabilité (conditionnelle) des valeurs qui apparait dans beaucoup de réseaux, il arrive souvent que deux états de décision distincts $\mathbf{s}_1$ et $\mathbf{s}_2$ considérés successivement pendant la recherche conduisent au même sous-problème, autrement dit $\mathcal{N} \mid \mathbf{s}_1$ et $\mathcal{N} \mid \mathbf{s}_2$ sont équivalents.

Dans une telle situation, plutôt que de compiler les deux réseaux, il est plus judicieux de ne compiler que $\mathcal{N} \mid \mathbf{s}_1$, puis d'enregistrer dans le cache une clé de $\mathcal{N} \mid \mathbf{s}_1$ associée à la racine N du nœud de sa représentation en MDDG et détecter que $\mathcal{N} \mid \mathbf{s}_2$ est équivalent à $\mathcal{N} \mid \mathbf{s}_1$ en cherchant dans le cache à chaque étape de la compilation. Ainsi, au lieu de calculer le graphe correspondant à $\mathcal{N} \mid \mathbf{s}_2$, il suffit de créer un arc pointant vers N. Par exemple, pour le réseau de contraintes utilisé comme exemple récurrent dans ce chapitre, la composante sur $\{X_3, X_4\}$ obtenue par décomposition dynamique est la même pour les états $\{\langle X_2, 0\rangle\}$ et $\{\langle X_2, 1\rangle\}$; nous n'avons donc pas besoin de la dupliquer.

Cependant, tester l'équivalence de deux réseaux de contraintes est un problème coNP-complet et un nombre exponentiel d'instances de ce problème devront être considérées un durant la compilation. Pour ces raisons, il est impossible de réaliser un *caching* en force brute dans lequel l'équivalence des réseaux serait effectivement testée. De ce fait, deux réseaux $\mathcal{N} \mid \mathbf{s}_1$ et $\mathcal{N} \mid \mathbf{s}_2$ sont considérés comme équivalents lorsqu'ils sont syntaxiquement identiques.

Une autre difficulté posée par le *caching* concerne la taille des entrées du cache, qui doivent rester aussi petites que possible. Pour notre mise en cache, nous enregistrons d'abord les domaines courants des variables du problème, c'est-à-dire $\mathbf{s}$ restreint à ses variables non affectées. Stocker l'ensemble des contraintes courantes serait bien trop gourmand en espace. Heureusement, c'est inutile dans la plupart

des cas. En effet, toute contrainte $C_j = (S_j, R_j)$ telle que $S_j \cap \mathit{single}(\mathbf{s}) = \varnothing$ n'a pas besoin d'être sauvegardée (étant donné qu'elle n'est pas affectée). De plus, aucune contrainte $C_j = (S_j, R_j)$ binaire du réseau donné en entrée n'a besoin d'être enregistrée à partir du moment où ce réseau est arc-cohérent. En effet, si aucune variable de $S_j = \{X_i, X_k\}$ n'a été instantiée, alors nous sommes dans le cas précédent. Si les deux variables X_i, X_k de S_j sont instantiées alors soit la contrainte est universelle, soit elle est incohérente et donc, il est inutile de la stocker (dans aucun des deux cas). Enfin, si une seule variable X_i de S_j est instantiée (disons à la valeur x_i) alors la projection sur la variable restante (non instantiée) X_k de la restriction R_j à $X_i = x_i$, coïncide avec la restriction de $\mathbf{s}$ à $\{X_k\}$ quand le réseau courant est arc-cohérent. Cela rappelle le traitement des clauses binaires dans `Dsharp`, qui n'ont pas besoin de figurer dans le cache du moment où la propagation unitaire a été effectuée [MMBH12].

De manière similaire, nous n'avons pas besoin d'enregistrer les contraintes *allDifferent* qui peuvent être considérées comme des conjonctions de contraintes binaires. Les contraintes restantes sont sauvegardées dans notre cache. Pour les contraintes représentées en intension, les variables instantiées dans $\mathbf{s}$ sont remplacées par leurs valeurs dans les prédicats, et on effectue une étape de simplification dans le but de réduire les représentations des contraintes. Les contraintes représentées en extension sont stockées explicitement. Finalement, pour chaque contrainte *weightedSum*, les variables instanciées dans $\mathbf{s}$ sont remplacées par leurs valeurs, les contraintes sont simplifiées et seul le terme constant en résultant a besoin d'être sauvegardé. Les contraintes de type *element* qui sont binaires n'ont pas besoin d'être sauvegardées. Pour les contraintes de types n-aires, on enregistre en cache $\{\langle X_i, x_i\rangle \in \mathbf{s} \mid X_i \in S_j\}$.

5.3.3 L'heuristique de choix de variable

Notre heuristique de choix de variable *bc* est basée sur une notion de centralité d'intermédiarité (*betweenness centrality*) utilisée en algorithmique des graphes [Bra08]. Étant donné un nœud $X_i \in \mathcal{X}$ dans un graphe $(\mathcal{X}, \mathcal{E})$, $bc(X_i)$ est donné par le nombre de plus courts chemins entre toute paire de nœuds passant par X_i. Dans notre cas, le graphe considéré $(\mathcal{X}, \mathcal{E})$ est le graphe primal du réseau de contraintes courant dans lequel les nœuds sont identifiés par les variables qu'ils étiquettent (et deux nœuds sont reliés par une arête quand les variables correspondantes sont utilisées dans une même contrainte). Formellement,

$$bc(X_i) = \Sigma_{X_j \neq X_i \neq X_k} \frac{\sigma_{X_i}(X_j, X_k)}{\sigma(X_j, X_k)}$$

où X_i, X_j et X_k sont des nœuds du réseau, $\sigma(X_j, X_k)$ est le nombre de plus courts chemins de X_j à X_k et $\sigma_{X_i}(X_j, X_k)$ est le nombre de ces chemins passant par X_i.

Il apparaît clairement qu'affecter en priorité les variables centrales du graphe primal $(\mathcal{X}, \mathcal{E})$ d'un réseau de contraintes favorise la génération de composantes connexes disjointes de taille similaire. Ceci permet la décomposition du réseau en plusieurs sous-réseaux indépendants (portant sur des ensembles disjoints de variables), de tailles proches, pouvant être compilés séparément et rassemblés par un nœud $\wedge$ dans la représentation cible en `MDDG`. Le calcul de la centralité d'intermédiarité de tous les nœuds dans $(\mathcal{X}, \mathcal{E})$ peut être effectué en $\mathcal{O}(n.p)$, où $n = \#(\mathcal{X})$ et $p = \#(\mathcal{E})$. En pratique, le calcul de $bc(X_i)$ pour chaque nœud X_i du graphe $(\mathcal{X}, \mathcal{E})$ d'un réseau de contraintes est suffisamment efficace pour être effectué dynamiquement, c'est-à-dire être calculé pour chaque réseau rencontré lors de la compilation.

Exemple 71 *Calcul de la centralité d'intermédiarité*

Soient N_1 et N_2 deux réseaux de contraintes schématisés respectivement par les figures 5.9 et 5.10. Pour le réseau de contraintes N_1, représentant l'exemple 69, X_2 est l'unique variable maximisant la valeur de bc. En effet, si nous suivons la formule 5.3.3 avec $i = 2$, le nombre de plus courts chemins entre chaque $j \neq k \neq 2$ est de 3. Nous avons à prendre en compte les chemins $X_1 - X_3$, $X_1 - X_4$ et $X_3 - X_4$. Le nombre de ces chemins passant par X_2 est donc de 2. Nous avons donc bc= $\frac{2}{3}$ pour ce

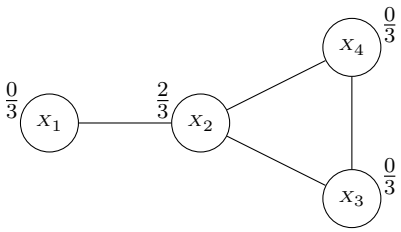


FIGURE 5.9 – Graphe primal du réseau de contraintes N_1

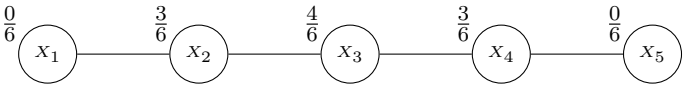


FIGURE 5.10 – Graphe primal du réseau de contraintes N_2

nœud. Pour chacun des trois autres nœud, nous pouvons facilement observer qu'il existe trois plus courts chemins entre les nœuds du graphe primal différents de celui considéré. Cependant, aucun d'entre eux ne passe par le nœud que nous considérons. Nous avons donc bc$= \frac{0}{3}$ *pour chacun de ces nœuds du graphe. L'heuristique basée sur la centralité d'intermédiarité proposera donc d'affecter en premier lieu la variable* X_2*. Nous remarquons qu'une fois la variable* X_2 *affectée, le graphe primal de* N_1 *peut être découpé en deux sous-graphes indépendants. D'un côté, le graphe contenant le nœud* X_1 *et de l'autre celui contenant les nœuds* X_3 *et* X_4*. Le nombre de composantes connexes a ainsi augmenté.*

Le même raisonnement s'applique pour le graphe primal du réseau de contraintes N_2*. Pour chaque nœud, nous avons 6 plus courts chemins différents entre chaque couple de* $\mid Var(N_2) \mid -1$ *variables du réseau de contraintes* N_2*. En particulier, le nœud* X_3 *figure dans 4 chemins. Nous avons les chemins* $X_1 - X_4$*,* $X_1 - X_5$*,* $X_2 - X_4$ *et* $X_2 - X_5$ *qui passent par le nœud* X_3*. Enfin, les chemins* $X_1 - X_2$ *et* $X_4 - X_5$*, qui, eux, ne passent pas par* X_3*. Donc la valeur retournée par l'heuristique bc pour* X_3 *est* $\frac{4}{6}$*.* X_3 *est l'unique variable maximisant la valeur de bc pour ce réseau de contraintes* N_2*. Nous remarquons qu'une fois cette variable instantiée,* N_2 *peut être découpé en deux sous-graphes indépendants de même taille. Le premier porte sur les variables* X_1 *et* X_2 *et le second sur les variables* X_4 *et* X_5*.*

5.3.4 L'algorithme de compilation

Nous pouvons maintenant décrire le compilateur cn2mddg que nous avons développé.

L'algorithme 8 présente le pseudo-code de ce compilateur. La compilation d'un réseau de contraintes $\mathcal{N}$ s'effectue en appellant cn2mddg sur ce réseau et l'état de décision initial $\mathbf{s} = \{\langle X_i, x_i\rangle \mid X_i \in \mathcal{X}, x_i \in D_i\}$. Tout d'abord (ligne 1), on établit l'arc-cohérence de $\mathcal{N}$ avec $\mathbf{s}$ (les valeurs des variables apparaissant dans $\mathbf{s}$ n'étant pas supportées par $\mathcal{N}$ sont supprimées de l'état). Pour des raisons d'efficacité, l'arc-cohérence est assurée au tout début (au premier appel) puis est maintenue dynamiquement à chaque fois qu'une nouvelle affectation élémentaire est considérée (ligne 13).

Ensuite, $\mathcal{N}$ est conditionné par l'état courant (ligne 2). Dans notre implémentation, le conditionnement $\mathcal{N} \mid \mathbf{s}$ à la ligne 2 est fait implicitement (nous le présentons ainsi pour des raisons de clarté). À chaque étape, seules les listes des variables non instantiées et des domaines courants sont mises à jour (pour des raisons d'efficacité, les contraintes ne sont jamais modifiées).

A la ligne 3, nous appelons un *solver* CSP afin de déterminer si $\mathcal{N}$ est cohérent. Nous avons développé notre propre *solver*, basé sur une recherche en profondeur avec retour en arrière chronologique,

Algorithme 8 : cn2mddg($\mathcal{N}$)

entrée : un réseau de contraintes $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, et un état de décision $\mathtt{s}$ sur une partie de $\mathcal{X}$
sortie : la racine N d'une représentation MDDG équivalente à $\mathcal{N} \mid \mathtt{s}$

```
1  s ← ac(N, s);
2  N ← N | s;
3  if unsat(N) then return leaf(⊥);
4  if #(X) = 0 then return leaf(⊤);
5  if cache(N) ≠ nil then return cache(N);
6  C ← removeUniversal(C);
7  CoCo ← connectedComponents(N);
8  N∧ ← ∅;
9  foreach Co ∈ CoCo do
10     Xi ← selectVariable(Co);
11     N∨ ← ∅;
12     foreach xi s.t. ⟨Xi, xi⟩ ∈ s do
13        N∨ ← N∨ ∪ cn2mddg(N, s[Co \ {Xi}] ∪ {⟨Xi, xi⟩});
14     N∧ ← N∧ ∪ dNode(Xi, N∨);
15 N ← aNode(N∧);
16 cache(N) ← N;
17 return N;
```

utilisant l'heuristique dirigée par les conflits (devenue standard) *dom/wdeg* [BHLS04]. L'arc-cohérence est maintenue à chaque point de choix. Un poids est associé à chaque contrainte C_j de $\mathcal{C}$, et est incrémenté chaque fois qu'un conflit impliquant C_j est détecté. Si $\mathcal{N}$ est incohérent, alors il est équivalent à la formule MDDG réduite à la feuille étiquetée par $\bot$, retournée par l'algorithme.

La ligne 4 concerne le dernier cas de base, quand toutes les variables de $\mathcal{X}$ ont été considérées. Dans ce cas, $\mathcal{N}$ est équivalent à la formule MDDG $\top$, retournée par l'algorithme.

On recherche alors si le réseau courant $\mathcal{N}$ a déjà été rencontré lors de la compilation (ligne 5). On utilise pour cela la fonction cache qui fait le lien entre les réseaux rencontrés et la racine du MDDG correspondant au réseau. Si $\mathcal{N}$ a déjà été trouvé durant la compilation, alors l'algorithme retourne simplement la racine du nœud de la formule MDDG correspondante.

Sinon on simplifie $\mathcal{N}$ en supprimant les contraintes universelles qu'elle contient (ceci est effectué par la fonction removeUniversal à la ligne 6).

Puis, on cherche les composantes connexes du réseau obtenu (ligne 7). La fonction connectedComponents retourne une partition *CoCo* de l'ensemble courant des variables $\mathcal{X}$ correspondant aux composantes connexes du graphe primal de $\mathcal{N}$ (un simple parcours en largeur d'abord du graphe permet de les déterminer). Ensuite, on considère chaque élément *Co* de *CoCo* successivement (ligne 9). *Co* est donc un ensemble de variables indépendant des autres éléments de *CoCo*. Chaque réseau $\mathcal{N}$ correspondant peut donc être compilé séparément, que l'on regroupe dans un ensemble de nœuds $N_\wedge$ préalablement initialisé à l'ensemble vide (ligne 8). Pour chaque réseau *Co*, l'état de décision courant $\mathtt{s}$ peut être réduit aux variables apparaissant dans *Co*. Une variable X_i est choisie grâce à la fonction selectVariable en ligne 10. Ensuite, en ligne 12, on considère successivement chacune des valeurs x_i du domaine courant de X_i. Plus précisément, pour chaque valeur x_i, on construit une affectation élémentaire correspondante $\{\langle X_i, x_i\rangle\}$. On conditionne le réseau par $\mathtt{s}$ que l'on a restreint aux variables de *Co* privées de X_i, auquel

on ajoute l'affectation $\{\langle X_i, x_i \rangle\}$; ce réseau est compilé récursivement (ligne 13). L'ensemble $N_\vee$ des nœuds de décision, initialisé à l'ensemble vide (ligne 11), est mis à jour (ligne 13). Lorsque toutes les valeurs x_i du domaine ont été considérées, on construit un nouveau nœud de décision étiqueté par X_i que l'on ajoute à l'ensemble des nœuds $N_\wedge$ (ligne 14). Une fois traité l'ensemble des éléments de *CoCo*, on rassemble les éléments de $N_\wedge$ en une conjonction sous forme d'un nœud $\wedge\, N$ grâce à la fonction aNode à la ligne 15. On associe ce nœud à l'entrée $\mathcal{N}$ du cache (ligne 16). Enfin, en ligne 17, nous retournons la racine du MDDG représentant $\mathcal{N}$.

L'exécution de l'algorithme cn2mddg est garantie de se terminer, puisqu'à chaque étape de récursion, au moins une variable du réseau initial $\mathcal{N}$ est instanciée. Par construction, les solutions du graphe MDDG obtenu comme résultat sont exactement les mêmes que celles du réseau $\mathcal{N}$ en entrée.

5.4 Expérimentations

Dans cette section, nous présentons les expérimentations réalisées pour évaluer notre compilateur cn2mddg et discutons des résultats obtenus.

Alors que la compilation de réseaux de contraintes est un sujet de recherche actif depuis des années (voir [Ami99, Vem92, AFM02, FV04, AFNP12, AFNP14]), il n'existe pas (à notre connaissance) de compilateurs adaptés aux réseaux portant sur des variables ayant des domaines non booléens et gérant des contraintes représentées en intension. En particulier, le compilateur AOMDD (disponible à l'adresse http://graphmod.ics.uci.edu/group/aomdd) suppose que toutes les contraintes du réseau donné en entrée sont représentées en extension par leurs n-uplets autorisés. Heureusement, un grand nombre d'encodages CNF pour les réseaux de contraintes existent (voir, entre autres [dK89, BB03, IM94, TTB11, Wal00, Gen02]). Ceci va nous permettre de comparer notre approche à une approche par compilation en Decision-DNNF précédée d'une transformation du réseau en une formule CNF.

5.4.1 Le cadre expérimental

Nous avons sélectionné 173 réseaux de contraintes issus de 15 familles différentes. Certaines instances contiennent des contraintes définies en extension, par leurs listes des n-uplets autorisés ou encore par leurs listes de n-uplets interdits. Pour d'autres instances, les contraintes sont données en intension.

Nous avons compilé chaque réseau de contraintes en une représentation en MDDG grâce au compilateur cn2mddg. Nous avons ensuite compilé ces mêmes réseaux de contraintes en une représentation en Decision-DNNF, en traduisant d'abord chaque réseau/instance en une formule CNF, puis en utilisant le compilateur Dsharp, qui produit une représentation en Decision-DNNF à partir d'une formule CNF.

L'efficacité de la compilation (i.e., le temps de compilation et la taille de la forme compilée produite) dépend fortement de la façon dont de l'instance est représentée. C'est pourquoi nous avons considéré deux encodages différents de réseaux de contraintes vers des formules CNF. Tout d'abord, le *sparse encoding*, qui consiste à encoder les domaines des variables, puis les contraintes suivant une méthode mixte (dans le but de minimiser le nombre de clauses générées, chaque contrainte est encodée en utilisant soit le *support encoding* soit le *conflict encoding*). Ensuite, le *log encoding*, qui utilise un nombre logarithmique de variables booléennes pour encoder les domaines des variables et qui encode les contraintes suivant le *conflict encoding*.

Certains traducteurs disponibles, comme Sugar [TTKB09] ou Azucar [TTB12], s'appuient sur des encodages qui ne préservent pas l'équivalence, et ne peuvent donc pas être utilisés tels quels dans notre cadre. En effet, ces approches sont orientées vers la résolution du problème de satisfaction. Par conséquent, elles cherchent à réduire le plus possible la taille de la formule générée, et cela au détriment

du maintien de l'équivalence logique (qui n'est pas indispensable dans la perspective du test de cohérence). Les formules obtenues ne sont qu'équivalentes pour la cohérence avec le réseau initial : le réseau est cohérent si et seulement si la formule CNF produite l'est aussi.

Pour chaque instance, nous avons calculé le temps de compilation (en secondes) et la taille de la formule compilée (le nombre d'arcs dans le graphe obtenu). Pour les approches basées sur une transformation en CNF, nous avons aussi calculé le temps de traduction et la taille de la formule CNF obtenue (le nombre de variables et de clauses). Nous avons lancé nos expérimentations sur un Quad-core Intel XEON X5550 avec 32Gio de mémoire. Pour chaque instance, nous avons fixé un temps limite (*time-out*) à 3600 secondes pour la phase de transformation en CNF et un espace mémoire limite de 8Gio pour stocker la formule CNF. De même, nous avons fixé un temps limite de 3600 secondes pour les phases de compilation et un espace mémoire limite de 8Gio pour calculer la représentation de la forme compilée.

5.4.2 Les résultats obtenus

Dans le temps et la mémoire alloués, le compilateur cn2mddg a réussi à compiler 131 instances sur les 173 testées. La compilation s'est terminée sur un *time-out* (TO) pour 32 instances, et sur un *memory-out* (MO) pour 10 instances. Nous pouvons observer un très grand écart de performances avec Dsharp qui n'a compilé que 83 instances quand le *sparse encoding* a été utilisé, et 61 instances lorsque le *log encoding* a été utilisé.

Plus en détails, la transformation en CNF a conduit à 27 *memory-out* avec le *sparse encoding* (respectivement 35 avec le *log encoding*). Sur les 146 (respectivement 138) instances restantes, la compilation des CNF obtenue en Decision-DNNF par Dsharp s'est terminée sur un *time-out* pour 24 instances (respectivement 21), et sur un *memory-out* pour 39 instances (respectivement 56).

Quel que soit l'encodage considéré, (*sparse encoding* ou *log encoding*, l'approche par transformation en CNF suivie d'une compilation en Decision-DNNF s'est révélée peu compétitive dans la plupart des cas. Ceci peut s'expliquer par le très grand nombre de variables booléennes apparaissant dans la formule CNF générée. Un autre explication de cette contre-performance est la perte d'information sur la structure du problème due au format CNF, comparé à la représentation en réseau de contraintes qui fournit beaucoup plus d'informations structurelles.

Notre compilateur cn2mddg s'est montré bien plus robuste puisqu'il a réussi à compiler beaucoup de réseaux de contraintes dont certains se sont montrés hors de portée des approches par transformation en CNF puis compilation en Decision-DNNF.

En particulier, chacune des 83 instances compilées avec succès par Dsharp (avec le *sparse encoding* en amont) se sont aussi montrées compilables en MDDG en utilisant cn2mddg. Les temps de compilation requis pour construire les MDDG des réseaux donnés en entrée sont souvent plus courts que les temps de compilation nécessaires pour créer les compilations en formules Decision-DNNF depuis les représentations CNF des réseaux donnés. Plus remarquablement encore, les tailles des formules compilées se révèlent toujours plus petites, parfois de plusieurs ordres de grandeur, quand le langage cible est MDDG comparé à celles obtenues quand le langage cible est Decision-DNNF.

On peut observer cet écart sur les figures 5.11 et 5.12, où chaque point représente l'une des 83 instances que Dsharp a réussi à compiler (avec le *sparse encoding* en amont). Dans ces schémas, si le point se trouve au dessus de la droite cela signifie que Dsharp est moins performant, que cn2mddg. S'il est en dessous, cela signifie que Dsharp est plus performant que cn2mddg et s'il est sur la droite, cela signifie que les deux approches ont donné les mêmes performances. Le temps nécessaire pour compiler (respectivement la taille de) la représentation en MDDG est donnée sur l'axe des abscisses et le temps nécessaire pour compiler (respectivement la taille de) la représentation en Decision-DNNF depuis la CNF encodée est donné sur l'axe des ordonnées (notons bien que toutes les échelles considérées sur les figures sont logarithmiques). Nous pouvons aussi remarquer que la différence de temps en faveur de

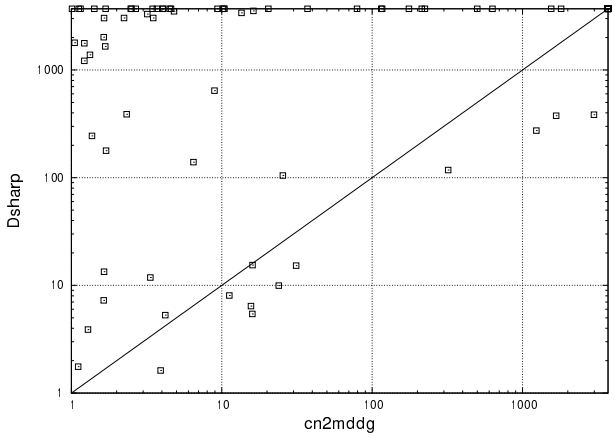


FIGURE 5.11 – Comparaison entre le temps de compilation en MDDG et le temps de compilation en Decision-DNNF.

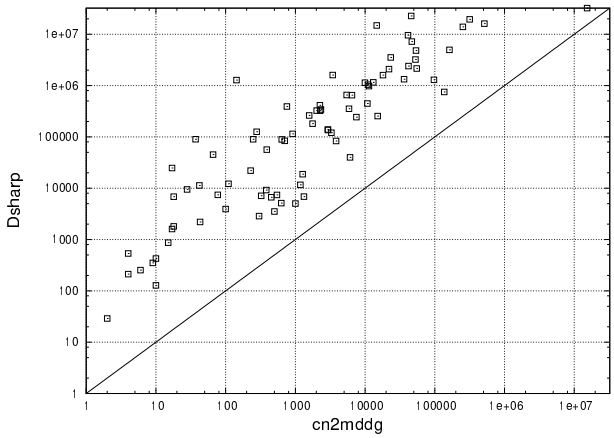


FIGURE 5.12 – Comparaison entre la taille des formes compilées en MDDG et la taille des formes compilées en Decision-DNNF.

cn2mddg aurait été bien plus grande encore si nous avions ajouté le temps nécessaire à l'encodage du réseau de contraintes en CNF au temps de compilation.

Le tableau 5.1 présente une sélection des résultats obtenus. Chaque ligne correspond à un réseau de contraintes $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ identifié par la colonne la plus à gauche. Les colonnes suivantes donnent respectivement le « type » de réseau de contraintes (en extension ou en intension), son nombre $\#\mathcal{X}$ de variables, son nombre $\#C$ de contraintes, l'arité maximale *maxA* des contraintes, la taille maximale *maxD* des domaines, un majorant *tw* de la largeur d'arbre de son graphe primal, le temps nécessaire pour compiler le réseau de contraintes en MDDG en utilisant cn2mddg, et la taille de la formule compilée. Le calcule du majorant de la largeur d'arbre du graphe primal est obtenu en utilisant QuickBB [GD04, GD] avec l'heuristique de choix de variable *random* et un *time-out* de 1800 secondes.

Pour les deux encodages en CNF que nous avons considérés, nous avons indiqué le nombre *#pv* de variables propositionnelles que contient la CNF, le nombre *#pcl* de clauses obtenues, le temps nécessaire pour compiler la CNF en Decision-DNNF en utilisant Dsharp, et la taille de la représentation en Decision-DNNF résultante.

Les résultats obtenus montrent que l'approche utilisant cn2mddg est plus compétitive que l'approche par transformation en CNF suivi d'une compilation enDecision-DNNF. La comparaison est, en effet,

en faveur de cn2mddg, que l'on considère comme critère de comparaison le nombre d'instances résolues, ou, ce qui est plus important, la taille des formules compilées. Les résultats montrent aussi que la compilation de réseaux de contraintes issus d'applications réelles, et possédant une complexité significative (souvent hors de portée de la compilation en *tree clustering* [DP89], vu la taille de leur domaine et la largeur d'arbre de leur graphe primal), est réalisable vers le langage MDDG.

Nous avons aussi réalisé une évaluation différentielle de chaque technique nouvelle utilisée dans cn2mddg afin de déterminer son apport. Soit dom/wdeg+noU la version de cn2mddg pour laquelle l'heuristique de choix de variable *dom/wdeg* est utilisée (à la place de *bc*), et pour laquelle les contraintes universelles ne sont pas gérées spécifiquement. Notre évaluation a montrée que dom/wdeg+noU a réussi à résoudre seulement 101 instances (sur 173) dans le temps et la mémoire allouées. De plus, le nombre d'instances (sur 101) pour lesquelles la taille de la représentation en MDDG obtenue en utilisant cn2mddg est inférieure à $p = \frac{1}{2}$ fois la taille de la représentation en MDDG obtenue en utilisant dom/wdeg+noU est de 35. A contrario, le nombre d'instances (sur 101) pour lesquelles la taille de la représentation en MDDG obtenue en utilisant dom/wdeg+noU est inférieure à $p = \frac{1}{2}$ fois la taille de la représentation en MDDG obtenue en utilisant cn2mddg est de 6. Les nombres correspondants si nous considérons une proportion de $p = \frac{1}{10}$, au lieu de $\frac{1}{2}$, tombe respectivement à 12 et à 0. Ainsi, pour plus de 10% des instances traitées, l'heuristique de choix de variable *bc* couplée à la gestion des contraintes universelles a conduit à une diminution de la taille de la forme compilée d'un ordre de grandeur.

5.5 Conclusion

Nous avons présenté dans ce chapitre le langage des graphes de décision décomposables multivalués (MDDG). Les propriétés de la carte de compilation offertes par ce langage ont été présentées. Nous avons montré en particulier que MDDG permet de compter le nombre de modèles en temps polynomial. Le langage MDDG vérifie en fait les mêmes requêtes et les mêmes transformations que le langage Decision-DNNF qu'il généralise.

Nous avons introduit un algorithme descendant cn2mddg pour la compilation de réseaux de contraintes à domaines finis, en graphes de décision décomposables multivalués. Parmi les différentes techniques qu'il incorpore, notre algorithme tire parti d'une méthode de *caching* spécifique, d'une nouvelle heuristique de choix de variable basée sur la centralité d'intermédiarité, et une gestion spécifique des contraintes universelles.

Notre large panel d'expérimentations a montré que cn2mddg permet de compiler des réseaux de contraintes qui ne peuvent pas être compilés en Decision-DNNF via un encodage préliminaire en CNF. De plus, nous avons montré que notre compilateur permet parfois de compiler des réseaux de contraintes qui n'ont pas pu être encodés en CNF. Nous avons aussi montré que cn2mddg permet typiquement de produire des représentations compilées de bien plus petites tailles que celles obtenues quand Decision-DNNF est ciblé. Ces expérimentations on aussi montré qu'utiliser notre nouvelle heuristique basée sur la centralité d'intermédiarité conduit sensiblement à améliorer les temps de compilation ainsi que les tailles des formules compilées.

	CN							
Name	type	$\#\mathcal{X}$	$\#C$	maxA	maxD	tw	time	size
rect-packing/rect-packingrpp07-true	I	1168	1273	8	23	15	31.22	5875
rect-packing/rect-packingrpp08-true	I	1584	1714	9	31	17	1234.72	162258
rect-packing/rect-packingrpp09-true	I	2196	2353	10	36	19	1673.33	514754
ghoulomb/ghoulomb3-4-6	I	3524	3543	16	37	?	164.95	74427
ghoulomb/ghoulomb3-4-5	I	2033	2051	11	26	31	15.17	5162
driver/normalized-driverlogw-08c-sat-ext	E	408	9321	2	11	92	15.63	2931
driver/normalized-driverlogw-08cc-sat-ext	E	408	9321	2	11	92	15.96	2931
scheduling/talent-concert	I	325	352	46	316	52	1277.21	404437
fapp/fapp18/normalized-fapp18-0350-8	I	350	2387	2	302	187	0.69	4243
fapp/fapp19/normalized-fapp19-0350-6	I	350	3114	2	802	130	79.34	1694146
costaArray/CostasArray10	I	110	338	4	19	23	10.39	13440
costaArray/CostasArray14	I	210	808	4	27	36	TO	-
photo/photophoto2	I	89	133	21	11	21	499.93	9564220
photo/photophoto1	I	75	102	18	9	18	10.13	210646
rlfap/normalized-scen4	I	680	3967	2	44	30	3.47	52226
rlfap/normalized-scen7-w1-f4	I	400	660	2	40	7	4.60	444483
radiation/radiation04	I	781	569	9	5180	33	-	MO
renault/normalized-renault-mod-32-ext	E	111	154	10	42	11	20.39	160238
renault/normalized-renault-mod-11-ext	E	111	149	10	42	10	16.22	41919
still-life/still-life7x7	I	690	803	50	50	49	1819.87	738478
configit/Aralia/edfpa15r	I	198	110	13	2	28	175.49	2044261
configit/Aralia/edfpa14q	I	505	194	22	2	34	TO	-
configit/Aralia/das9207	I	600	324	8	2	15	8.94	45853

	CNF - sparse mixed encoding				CNF - log conflict encoding			
Name	#pv	#pcl	time	size	#pv	#pcl	time	size
rect-packing/rect-packingrpp07-true	8709	110661	15.26	353048	2378	70087	37.70	976110
rect-packing/rect-packingrpp08-true	17724	248417	273.73	4973120	3225	147307	1139.74	7938679
rect-packing/rect-packingrpp09-true	37044	593518	375.66	16118647	4466	392657	TO	-
ghoulomb/ghoulomb3-4-6	MO	MO	MO	MO	MO	MO	MO	MO
ghoulomb/ghoulomb3-4-5	MO	MO	MO	MO	MO	MO	MO	MO
driver/normalized-driverlogw-08c-sat-ext	9528	62825	6.42	139306	1050	46081	32.78	499796
driver/normalized-driverlogw-08cc-sat-ext	9528	62825	5.43	136501	1050	46081	23.63	425617
scheduling/talent-concert	MO	MO	MO	MO	MO	MO	MO	MO
fapp/fapp18/normalized-fapp18-0350-8	9199429	63582486	-	MO	2810	52179077	-	MO
fapp/fapp19/normalized-fapp19-0350-6	166130802	867243022	-	MO	MO	MO	MO	MO
costaArray/CostasArray10	149564	841930	TO	-	540	3606946	TO	-
costaArray/CostasArray14	988671	5568047	TO	-	1036	34687218	-	MO
photo/photophoto2	685555	14326576	TO	-	204	10923133	-	MO
photo/photophoto1	73095	1328839	TO	-	172	1117281	TO	-
rlfap/normalized-scen4	915553	4875002	-	MO	4060	3058032	TO	-
rlfap/normalized-scen7-w1-f4	102556	683178	-	MO	2392	522995	-	MO
radiation/radiation04	MO	MO	MO	MO	MO	MO	MO	MO
renault/normalized-renault-mod-32-ext	222582	1755876	TO	-	286	138124077	-	MO
renault/normalized-renault-mod-11-ext	223718	1762294	3538.01	2399273	286	138117804	-	MO
still-life/still-life7x7	MO	MO	MO	MO	MO	MO	MO	MO
configit/Aralia/edfpa15r	396	24710	-	MO	396	24710	-	MO
configit/Aralia/edfpa14q	1010	5793030	TO	-	1010	5793030	TO	-
configit/Aralia/das9207	1200	3894	642.68	22707412	1200	3894	97.86	9937506

TABLE 5.1 – Une sélection de résultats expérimentaux.

Conclusion générale

1 Rappel des contributions

Dans cette thèse, nous avons proposé et étudié une approche par compilation de connaissances dans l'objectif de compter plus efficacement le nombre de modèles de représentations de nature booléenne. Nous nous sommes intéressés à la compilation de représentations issues de deux formalismes de représentation, la logique propositionnelle et les réseaux de contraintes.

Nous avons présenté dans les deux premiers chapitres les deux formalismes de représentation sur lesquels nous nous sommes focalisés, et nous avons décrit les principales méthodes de recherche arborescente de solutions qui ont été développées pour les représentations de ces formalismes. Dans le chapitre suivant, nous avons rappelé les principes qui régissent la compilation de connaissances et le concept de carte de compilation. Nous avons également présenté plusieurs langages cibles pour la compilation de formules propositionnelles.

La suite du mémoire a été essentiellement consacrée aux contributions apportées par cette thèse.

Nous avons, dans le chapitre 4, proposé de nouveaux langages cibles pour la compilation de formules propositionnelles. En particulier, les langages EDT des arbres de décision étendus, ADT des arbres de décision affine et $\texttt{EADT}_\wedge$ des arbres de décision affine étendus, ont été introduits. Ces langages ont été analysés selon les critères mis en œuvre dans la carte de compilation. Nous avons ainsi déterminé des requêtes et des transformations réalisables en temps polynomial à partir de représentations dans ces langages, et également montré que d'autres requêtes et transformations ne le sont pas sauf si P = NP. Nous avons également comparé l'efficacité spatiale relative de ces langages à celles d'autres langages cibles existants. Il ressort de cette étude que le langage $\texttt{EADT}_\wedge$ est intéressant lorsque la requête du comptage de modèles est une question cruciale. En effet, les requêtes pour lesquelles $\texttt{EADT}_\wedge$ et le langage d-DNNF possèdent des algorithmes en temps polynomial sont les mêmes. Nous avons aussi montré que le sous-ensemble ADT de $\texttt{EADT}_\wedge$ satisfait toutes les requêtes et offre les mêmes transformations que le langage $\texttt{OBDD}_<$. De plus, $\texttt{OBDD}_<$ n'est pas au moins aussi succinct que le langage des arbres de décision affine, ce qui place ADT comme un possible challenger de $\texttt{OBDD}_<$.

Le compilateur du langage CNF vers $\texttt{EADT}_\wedge$ que vous avons développé a également été décrit dans ce chapitre. Les expérimentations réalisées ont montré que ce langage est réellement attractif en pratique. En effet, il se révèle très compétitif face aux algorithmes existants de l'état de l'art pour le comptage de modèles, comme le compteur de modèles direct Cachet et le compilateur **c2d** ciblant le langage Decision-DNNF. Pour certaines instances, la compilation vers le langage $\texttt{EADT}_\wedge$ suivie du comptage de modèles s'est même montré plus efficace que le comptage de modèles direct.

Dans le chapitre 5, nous avons défini un langage cible pour la compilation des réseaux de contraintes, le langage MDDG des graphes de décision décomposables multivalués. Nous avons montré que ce langage offre la requête **CT**, c'est-à-dire qu'il existe un algorithme en temps polynomial pour compter les modèles d'un graphe MDDG. Le langage MDDG offre en fait les mêmes requêtes et transformations que le langage Decision-DNNF dont il constitue une généralisation. MDDG se montre donc un langage cible très intéressant pour la compilation de réseaux de contraintes.

Le compilateur du langage des réseaux de contraintes vers MDDG que vous avons développé a également été décrit dans ce chapitre. Ce compilateur repose sur plusieurs ingrédients standard (comme la recherche de composantes connexes) et d'autres qui sont originaux (l'utilisation d'une méthode spécifique de *caching*, d'une heuristique orientée pour la compilation, et une gestion spécifique des contraintes universelles). Ces ingrédients ont permis d'atteindre des performances tout à fait acceptables. Ainsi, les expérimentations réalisées, décrites dans le chapitre, ont montré que notre compilateur permet de compiler des réseaux de contraintes qui ne peuvent pas être compilés en Decision-DNNF via un encodage CNF préliminaire. Elles ont même montré que notre compilateur permet de compiler des réseaux de contraintes qui n'ont parfois pas pu être encodés en CNF. Nous avons aussi montré que la compilation sous forme de graphes MDDG permet d'obtenir des représentations de tailles significativement plus petites que celles obtenues quand le langage Decision-DNNF est ciblé.

2 Perspectives

Les différentes contributions apportées ouvrent de nombreuses perspectives pour étendre nos travaux.

D'un point de vue théorique, il serait intéressant de déterminer d'autres langages cibles de compilation intégrant des nœuds de décision affine. Par ailleurs, bien que les langages ADT et $\text{EADT}_\wedge$ soient des langages cibles compétitifs pour la requête du comptage de modèles, ils ne permettent pas le comptage de modèles pondérés en temps polynomial (et il ne peut exister un algorithme en temps polynomial pour réaliser cette requête sur les formules ADT ou $\text{EADT}_\wedge$ sauf si P = NP). Le problème du comptage de modèles pondérés consiste à calculer, étant donnée une application associant un poids à chaque littéral possible, la somme des poids des modèles d'une formule, où le poids d'un modèle est le produit des poids des littéraux que le modèle satisfait. Le problème du calcul de la probabilité d'un événement dans un réseau bayésien peut être réduit en temps polynomial au problème du comptage de modèles pondérés (voir par exemple [BKLM16]), ce qui explique l'importance de cette requête. Or, le problème du comptage de modèles pondérés d'une formule affine a été prouvé **#P**-complet [DGJ09], et comme chaque formule affine peut être traduite en temps polynomial en une formule ADT (donc $\text{EADT}_\wedge$), ce résultat négatif s'étend aux langages ADT et $\text{EADT}_\wedge$. Néanmoins, le comptage de modèles pondérés est réalisable en temps polynomial à partir de formules $\text{EADT}_\wedge$ lorsque les clauses affines étiquetant les nœuds de décision portent sur au plus deux variables. En outre, il est possible de réduire le problème du comptage de modèles pondérés au problème standard de comptage de modèles (**CT**), comme montré dans [CFMV15]. Il serait donc intéressant d'exploiter ces pistes et d'évaluer l'intérêt que pourraient offrir les langages ADT et $\text{EADT}_\wedge$ pour compiler des réseaux bayésiens.

Différentes pistes existent pour améliorer notre compilateur ciblant le langage $\text{EADT}_\wedge$. En particulier, il serait intéressant de profiter plus encore de certaines techniques de pré-traitement développées pour le problème SAT (comme celles décrites dans [PHS08, JHB12]) afin de simplifier la formule CNF en entrée avant de la compiler. De plus, il pourrait être utile d'exploiter l'élimination de Gauss pour traiter plus efficacement (voir par exemple [Li03, Che07, SNC09]) les instances contenant des sous-problèmes correspondant à des clauses affines, comme celles données dans [CK95, Can06]. Finalement, tenir compte d'autres heuristiques pour choisir les clauses affines de branchement (par exemple, utiliser un critère basé sur la mesure d'information mutuelle) pourrait rendre plus efficace la compilation.

Il serait aussi intéressant de généraliser le langage MDDG et de définir ainsi d'autres langages cibles pour la compilation de réseaux de contraintes. Une piste est d'étendre la notion de nœud de décision affine au cas des variables non binaires comme celles rencontrées dans les réseaux de contraintes. Cela conduirait à rapprocher les deux contributions principales de cette thèse. Enfin, une perspective importante est d'étendre notre approche à la compilation de réseaux de contraintes pondérées, de façon à pouvoir compiler, entre autres, des modèles graphiques probabilistiques (e.g. réseaux bayésiens ou markoviens).

Bibliographie

[AFM02] J. AMILHASTRE, H. FARGIER, et P. MARQUIS. « Consistency restoration and explanations in dynamic CSPs Application to configuration ». *Artificial Intelligence*, 135(1-2) :199–234, 2002. 5.4

[AFNP12] J. AMILHASTRE, H. FARGIER, A. NIVEAU, et C. PRALET. « Compiling CSPs : A Complexity Map of (Non-Deterministic) Multivalued Decision Diagrams ». Dans *Proc. of ICTAI'12*, pages 1–8, 2012. 5.4

[AFNP14] J. AMILHASTRE, H. FARGIER, A. NIVEAU, et C. PRALET. « Compiling CSPs : A Complexity Map of (Non-Deterministic) Multivalued Decision Diagrams ». *International Journal on Artificial Intelligence Tools*, 23(4), 2014. 5.2, 5.4

[Ake78] S.B. AKERS. « Binary decision diagrams ». *IEEE Transactions on Computers*, C-27(6) :509–516, 1978. 2

[Ami99] J. AMILHASTRE. « *Représentation par automate de l'ensemble des solutions de problèmes de satisfaction de contraintes* ». PhD thesis, Université des Sciences et Techniques du Languedoc, Montpellier, 1999. 5.4

[AS09] Gilles AUDEMARD et Laurent SIMON. « Predicting Learnt Clauses Quality in Modern SAT Solver ». Dans *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, jul 2009. 2.1.5

[BB03] Olivier BAILLEUX et Yacine BOUFKHAD. « Efficient CNF Encoding of Boolean Cardinality Constraints ». Dans *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, pages 108–122, 2003. 1.3.1, 5.4

[BBH+09] A. BIERE, A. BIERE, M. HEULE, H. van MAAREN, et T. WALSH. *Handbook of Satisfiability : Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009. 2.1, 2.1.4

[Ber01] Daniel Le BERRE. « Exploiting the real power of unit propagation lookahead ». *Electronic Notes in Discrete Mathematics*, 9 :59–80, 2001. 2.1.2

[Bes06] Christian BESSIERE. Constraint Propagation. Dans *Handbook of Constraint Programming*, pages 29–83. Elsevier, 2006. 2.2.3, 2.2.3

[BHLS04] F. BOUSSEMART, F. HEMERY, Ch. LECOUTRE, et L. SAÏS. « Boosting Systematic Search by Weighting Constraints ». Dans *Proc. of ECAI'04*, pages 146–150, 2004. 6, 17

[Bie08a] Armin BIERE. « Adaptive Restart Strategies for Conflict Driven SAT Solvers ». Dans Hans KLEINE BÜNING et Xishun ZHAO, éditeurs, *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 de *Lecture Notes in Computer Science*, pages 28–33. Springer Berlin / Heidelberg, 2008. 2.1.5

[Bie08b] Armin BIERE. « PicoSAT essentials ». *Journal on Satisfiability, Boolean Modeling and Computation*, 4(75-97) :45, 2008. 2.1.5

[BJS97] Roberto J. BAYARDO JR. et Robert C. SCHRAG. « Using CSP Look-Back Techniques to Solve Real-World SAT Instances ». pages 203–208, Providence (Rhode Island, USA), juillet 1997. 2.1.3

[BKLM16] A. BART, F. KORICHE, J.-M. LAGNIEZ, et P. MARQUIS. « An Improved CNF Encoding Scheme for Probabilistic Inference ». Dans *Proc. of ECAI'16*, pages 613–621, 2016. 2

[BMFL99] Christian BESSIÈRE, Pedro MESEGUER, Eugene C. FREUDER, et Javier LARROSA. « On Forward Checking for Non-binary Constraint Satisfaction ». Dans *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, pages 88–102, 1999. 2.2.3

[Boo54] George BOOLE. *Les lois de la pensée*. Mathesis, 1854. 2

[BPW03] J. Christopher BECK, Patrick PROSSER, et Richard J. WALLACE. « Toward Understanding Variable Ordering Heuristics for Constraint Satisfaction Problems ». Dans *In : procedings of the fourfteenth irish artificial intelligence and cognitive science conference-AIC'03*, pages 11–16, 2003. 6

[BPW04] J. Christopher BECK, Patrick PROSSER, et Richard J. WALLACE. « Variable Ordering Heuristics Show Promise ». Dans *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 711–715, 2004. 6

[Bra08] U. BRANDES. « On variants of shortest-path betweenness centrality and their generic computation ». *Social Networks*, 30(2) :136–145, 2008. 5.3.3

[Bry92] Randal E. BRYANT. « Symbolic Boolean manipulation with ordered binary-decision diagrams ». *ACM Computing Surveys*, 24(3) :293–318, 1992. 2

[BvR95] Fahiem BACCHUS et Paul van RUN. « Dynamic Variable Ordering in CSPs ». Dans Ugo MONTANARI et Francesca ROSSI, éditeurs, *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, volume 976 de *Lecture Notes in Computer Science*, pages 258–275. Springer, 1995. 6

[Can06] Ch. De CANNIÈRE. « Trivium : A Stream Cipher Construction Inspired by Block Cipher Design Principles ». Dans *Proc. of ISC'06*, pages 171–186, 2006. 2

[CD98] M. CADOLI et F.M. DONINI. « A survey on knowledge compilation ». *AI Communications*, 10(3–4) :137–150, 1998. 3

[CDLS02] M. CADOLI, F.M. DONINI, P. LIBERATORE, et M. SCHAERF. « Preprocessing of intractable problems ». *Information and Computation*, 176(2) :89–120, 2002. 3

[CFMV15] Supratik CHAKRABORTY, Dror FRIED, Kuldeep S. MEEL, et Moshe Y. VARDI. « From Weighted to Unweighted Model Counting ». Dans *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 689–695, 2015. 2

[CGS01] Hubie CHEN, Carla GOMES, et Bart SELMAN. « Formal Models of Heavy-Tailed Behavior in Combinatorial Search ». Dans Toby WALSH, éditeur, *Principles and Practice of Constraint Programming - CP 2001*, volume 2239 de *Lecture Notes in Computer Science*, pages 408–421. Springer Berlin / Heidelberg, 2001. 2.1.5

[Che07] J. CHEN. « XORSAT : An Efficient Algorithm for the DIMACS 32-bit Parity Problem ». *CoRR*, abs/cs/0703006, 2007. 2

[CK95] J.M. CRAWFORD et M.J. KEARNS. « The Minimal Disagreement Parity Problem as a Hard Satisfiability Problem ». Rapport technique, 1995. 2

[Coo71] Stephen A. COOK. « The Complexity of Theorem-Proving Procedures ». Dans *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971. 1.1.3, 2.1

[Dar01] A. DARWICHE. « Decomposable negation normal form ». *Journal of the ACM*, 48(4) :608–647, 2001. 2, 3

[Dar04] Adnan DARWICHE. « New Advances in Compiling CNF into Decomposable Negation Normal Form ». Dans *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332, 2004. 2.3.1, 2.3.3

[DB97] Romuald DEBRUYNE et Christian BESSIÈRE. « Some Practicable Filtering Techniques for the Constraint Satisfaction Problem ». Dans *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 412–417, 1997. 2.2.3

[DFW02] Lyndon DRAKE, Alan M. FRISCH, et Toby WALSH. « Automatic Generation of Implied Clauses for SAT ». Dans *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, page 783, 2002. 1.3.2

[DGJ09] Martin E. DYER, Leslie Ann GOLDBERG, et Mark JERRUM. « The Complexity of Weighted Boolean CSP ». *SIAM J. Comput.*, 38(5) :1970–1986, 2009. 2

[dK89] J. de KLEER. « A Comparison of ATMS and CSP Techniques ». Dans *Proc. of IJCAI'89*, pages 290–296, 1989. 1.3.1, 5.4

[DLL62a] M. DAVIS, G. LOGEMANN, et D. LOVELAND. « A machine program for theorem-proving ». *Communications of the ACM*, 5(7) :394–397, 1962. 2

[DLL62b] M. DAVIS, G. LOGEMANN, et D. LOVELAND. « A machine program for theorem-proving ». *Journal of the ACM*, 5(7) :394–397, July 1962. 2.1.1

[DM89] Rina DECHTER et Itay MEIRI. « Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems ». Dans N. S. SRIDHARAN, éditeur, *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, pages 271–277. Morgan Kaufmann, 1989. 6

[DM02] A. DARWICHE et P. MARQUIS. « A knowledge compilation map ». *Journal of Artificial Intelligence Research*, 17 :229–264, 2002. 3, 3.1, 3.1.1, 3.1.1, 3.1.2, 3.2, 3.2.1, 4.2, 4.2.1

[DP87] Rina DECHTER et Judea PEARL. « Network-based heuristics for constraint-satisfaction problems ». *Artificielle Intelligence*, 34 :1–38, December 1987. 4

[DP89] R. DECHTER et J. PEARL. « Tree Clustering for Constraint Networks ». *Artificial Intelligence*, 38(3) :353–366, 1989. 5.4.2

[ES03] N. EÉN et N. SÖRENSSON. An Extensible SAT-solver. Dans *Proc. of SAT'03*, pages 502–518. 2003. 6

[FM06] H. FARGIER et P. MARQUIS. « On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae ». Dans *Proc. of AAAI'06*, pages 42–47, 2006. 5.2

[FM08a] H. FARGIER et P. MARQUIS. « Extending the Knowledge Compilation Map : Closure Principles ». Dans *Proc. of ECAI'08*, pages 50–54, 2008. 3.2

[FM08b] H. FARGIER et P. MARQUIS. « Extending the Knowledge Compilation Map : Krom, Horn, Affine and Beyond ». Dans *Proc. of AAAI'08*, pages 442–447, 2008. 3.2, 10

[FM14] H. FARGIER et P. MARQUIS. « Disjunctive closures for knowledge compilation ». *Artificial Intelligence*, 216 :129–162, 2014. 4.2.3

[Fre85] Eugene C. FREUDER. « A Sufficient Condition for Backtrack-Bounded Search ». *J. ACM*, 32(4) :755–761, 1985. 2.2.3

[FV04] H. FARGIER et M.-C. VILAREM. « Compiling CSPs into tree-driven automata for interactive solving ». *Constraints*, 9 :263–287, 2004. 5.4

[Gal77] Z. GALIL. « On the complexity of regular resolution and the Davis-Putnam Procedure ». *Theoretical Computer Science*, 4 :23–46, 1977. 2

[GB65] Solomon W. GOLOMB et Leonard D. BAUMERT. « Backtrack Programming ». *J. ACM*, 12 :516–524, October 1965. 2

[GD] Vibhav GOGATE et Rina DECHTER. « QuickBB. ». http://www.hlt.utdallas.edu/~vgogate/quickbb.html. 5.4.2

[GD04] Vibhav GOGATE et Rina DECHTER. « A Complete Anytime Algorithm for Treewidth ». Dans *UAI '04, Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, Banff, Canada, July 7-11, 2004*, pages 201–208, 2004. 5.4.2

[Gen02] I. P. GENT. « Arc Consistency in SAT ». Dans *Proc. of ECAI'02*, pages 121–125, 2002. 1.3.2, 1.3.2, 5.4

[GFHT90] Matthew L. GINSBERG, Michael FRANK, Michael P. HALPIN, et Mark C. TORRANCE. « Search Lessons Learned from Crossword Puzzles ». Dans Howard E. SHROBE, Thomas G. DIETTERICH, et William R. SWARTOUT, éditeurs, *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, July 29 - August 3, 1990, 2 Volumes.*, pages 210–215. AAAI Press / The MIT Press, 1990. 6

[GHSS07] C. P. GOMES, J. HOFFMANN, A. SABHARWAL, et B. SELMAN. « From Sampling to Model Counting ». Dans *Proc. of IJCAI'07*, pages 2293–2299, 2007. 2.3.2

[GKPS95] G. GOGIC, H.A. KAUTZ, Ch.H. PAPADIMITRIOU, et B. SELMAN. « The comparative linguistics of knowledge representation ». Dans *Proc. of IJCAI'95*, pages 862–869, 1995. 3.1.3

[GMP+96] Ian P. GENT, Ewan MACINTYRE, Patrick PROSSER, Barbara M. SMITH, et Toby WALSH. « An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem ». Dans Eugene C. FREUDER, éditeur, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 de *Lecture Notes in Computer Science*, pages 179–193. Springer, 1996. 6

[GOMS04] Éric GRÉGOIRE, Richard OSTROWSKI, Bertrand MAZURE, et Lakhdar SAIS. « Automatic Extraction of Functional Dependencies ». Dans *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, pages 122–132, 2004. 2.1.2

[GSCK00] Carla P. GOMES, Bart SELMAN, Nuno CRATO, et Henry KAUTZ. « Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems ». *J. Autom. Reason.*, 24 :67–100, February 2000. 2.1.5

[GSS09] Carla P. GOMES, Ashish SABHARWAL, et Bart SELMAN. Model Counting. Dans *Handbook of Satisfiability*, pages 633–654. 2009. 2.3

[GW94] Ian P. GENT et Tobby WALSH. « Easy problems are sometimes hard ». *Artificial Intelligence*, 70 :335–345, 1994. 2.1.5

[HD07] J. HUANG et A. DARWICHE. « The Language of Search ». *Journal of Artificial Intelligence Research*, 29 :191–219, 2007. 3, 5.3

[HE80] Robert M. HARALICK et Gordon L. ELLIOTT. « Increasing Tree Search Efficiency for Constraint Satisfaction Problems ». *Artif. Intell.*, 14(3) :263–313, 1980. 6, 2.2.3

[HJS10] Youssef HAMADI, Saïd JABBOUR, et Lakhdar SAÏS. « LySAT : Solver description ». SATRACE, solver description, 2010. 2.1.5

[HM05] Joey HWANG et David G. MITCHELL. « 2-Way vs. d-Way Branching for CSP ». Dans Peter van BEEK, éditeur, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 de *Lecture Notes in Computer Science*, pages 343–357. Springer, 2005. 6

[HO05] Tudor HULUBEI et Barry O'SULLIVAN. « Search Heuristics and Heavy-Tailed Behaviour ». Dans *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pages 328–342, 2005. 6

[IM94] K. IWAMA et S. MIYAZAKI. « SAT-Variable Complexity of Hard Combinatorial Problems ». Dans *Proc. of IFIP World Computer Congress'94*, pages 253–258, 1994. 1.3.1, 5.4

[JHB12] M. JÄRVISALO, M. HEULE, et A. BIERE. « Inprocessing Rules ». Dans *Proc. of IJCAR'12*, pages 355–370, 2012. 2

[JW90] R. J. JEROSLOW et J. WANG. « Solving propositional satisfiability problems ». *Annals of Mathematics and Artificial Intelligence*, 1 :167–188, 1990. 2

[KLMT13] Frédéric KORICHE, Jean-Marie LAGNIEZ, Pierre MARQUIS, et Samuel THOMAS. « Knowledge Compilation for Model Counting : Affine Decision Trees ». Dans *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 947–953, 2013. 4, 2

[KLMT15] Frédéric KORICHE, Jean-Marie LAGNIEZ, Pierre MARQUIS, et Samuel THOMAS. « Compiling Constraint Networks into Multivalued Decomposable Decision Graphs ». Dans Qiang YANG et Michael WOOLDRIDGE, éditeurs, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 332–338. AAAI Press, 2015. 5

[LA97] Chu Min LI et ANBULAGAN. « Heuristics Based on Unit Propagation for Satisfiability Problems ». Dans *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 366–371, 1997. 2.1.4

[Lag11] Jean-Marie LAGNIEZ. « *Satisfiabilité propositionnelle et raisonnement par contraintes : modèles et algorithmes* ». PhD thesis, CRIL, Lens, France, Dec 2011. 2.1.4

[Lau78] Jean-Louis LAURIÈRE. « A Language and a Program for Stating and Solving Combinatorial Problems ». *Artif. Intell.*, 10(1) :29–127, 1978. 1.2.1

[LBH04] Christophe LECOUTRE, Frédéric BOUSSEMART, et Fred HEMERY. « Backjump-Based Techniques versus Conflict-Directed Heuristics ». Dans *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, pages 549–557. IEEE Computer Society, 2004. 6

[LH05] Chu LI et Wen HUANG. « Diversification and Determinism in Local Search for Satisfiability ». Dans *Theory and Applications of Satisfiability Testing (SAT)*, pages 158–172. Springer, 2005. 2.1.5

[Li03] C. LI. « Equivalent literal propagation in the DLL procedure ». *Discrete Applied Mathematics*, 130(2) :251–276, 2003. 2

[Lib00] Paolo LIBERATORE. « On the complexity of choosing the branching literal in DPLL ». *Artif. Intell.*, 116(1-2) :315–326, 2000. 2.1.4, 6

[LM14] J.-M. LAGNIEZ et P. MARQUIS. « Preprocessing for Propositional Model Counting ». Dans *Proc. of AAAI'14*, pages 2688–2694, 2014. 6

[LSZ93] Michael LUBY, Alistair SINCLAIR, et David ZUCKERMAN. « Optimal speedup of Las Vegas algorithms ». *Information Processing Letters*, 47(4) :173–180, 1993. 2.1.5

[Mac77] Alan K. MACKWORTH. « Consistency in Networks of Relations ». *Artif. Intell.*, 8(1) :99–118, 1977. 2.2.3

[MD06] R. MATEESCU et R. DECHTER. « Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs) ». Dans *Proc. of CP'06*, pages 329–343, 2006. 5.2

[MDM08] R. MATEESCU, R. DECHTER, et R. MARINESCU. « AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Graphical Models ». *Journal of Artificial Intelligence Research*, 33 :465–519, 2008. 5.2

[MMBH12] Ch.J. MUISE, Sh.A. MCILRAITH, J.Ch. BECK, et E.I. HSU. « Dsharp : Fast d-DNNF Compilation with sharpSAT ». Dans *Proc. of AI'12*, pages 356–361, 2012. 2.3.3, 8, 5.3.2

[MMZ+01a] Matthew W. MOSKEWICZ, Conor F. MADIGAN, Ying ZHAO, Lintao ZHANG, et Sharad MALIK. « Chaff : Engineering an Efficient SAT Solver ». Dans *Proc. of DAC '01*, pages 530–535, 2001. 2.1

[MMZ+01b] M.W. MOSKEWICZ, C.F. MADIGAN, Y. ZHAO, L. ZHANG, et S. MALIK. « Chaff : engineering an efficient SAT solver ». Dans *Proc. of DAC'01*, pages 530–535, 2001. 2.1.1, 2.1.3, 2.1.5, 6, 6

[Mon74] Ugo MONTANARI. « Networks of constraints : Fundamental properties and applications to picture processing ». *Inf. Sci.*, 7 :95–132, 1974. 2.2.3

[MSS96] J.oão P. MARQUES-SILVA et Karem A. SAKALLAH. « Conflict Analysis in Search Algorithms for Propositional Satisfiability ». Rapport technique RT/4/96, Grupo ALGOS : Instituto de Engenharia de Sistemas e Computadores, Lisboa (Portugal), mai 1996. 2.1.1, 2.1.3

[NBR] Mats Carlsson NICOLAS BELDICEANU et Jean-Xavier RAMPON. « Global Constraint Catalog. ». http://sofdem.github.io/gccat/. 1.2.1

[NGPH06] Alexandre NADEL, Maon GORDON, Amit PATI, et Ziad HANA. « Eureka-2006 SAT solver ». SAT-Race, solver description, 2006. 2.1.5

[Nov03] Yakov NOVIKOV. « Local Search for Boolean Relations on the Basis of Unit Propagation ». Dans *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10810–10815, 2003. 2.1.2

[OD14] Umut OZTOK et Adnan DARWICHE. « On Compiling CNF into Decision-DNNF ». Dans *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 42–57, 2014. 2, 11, 5.2

[PD07] Knot PIPATSRISAWAT et Adnan DARWICHE. « A Lightweight Component Caching Scheme for Satisfiability Solvers ». Dans *SAT*, pages 294–299, 2007. 2.1.5

[PD08] K. PIPATSRISAWAT et A. DARWICHE. « New Compilation Languages based on Structured Decomposability ». Dans *Proc. of AAAI'08*, pages 517–522, 2008. 11, 4.2.3

[PD10] K. PIPATSRISAWAT et A. DARWICHE. « A Lower Bound on the Size of Decomposable Negation Normal Form ». Dans *Proc. of AAAI'10*, 2010. 4.2.3

[PHS08] Cédric PIETTE, Youssef HAMADI, et Lakhdar SAIS. « Vivifying Propositional Clausal Formulae ». Dans *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, pages 525–529, 2008. 2.1.2, 2

[PJ10] Justyna PETKE et Peter JEAVONS. « Local Consistency and SAT-Solvers ». Dans *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pages 398–413, 2010. 1.3.2

[Pro93] Patrick PROSSER. « Hybrid algorithms for the constraint satisfaction problems ». *Computational Intelligence*, 9(3) :268–299, 1993. 2.1.3

[Qui50] William V. QUINE. *Methods of logics*. Henry Holt, 1950. 2

[RL09] O. ROUSSEL et Ch. LECOUTRE. « XML Representation of Constraint Networks : Format XCSP 2.1 ». Rapport technique, Computing Research Repository (CoRR) abs/0902.2362, feb 2009. 5.3

[Rob65] J.A. ROBINSON. « A machine-oriented logic based on the resolution principle ». *Journal of the Association for Computing Machinery*, 12 :23–41, 1965. 2

[RPD90] Francesca ROSSI, Charles PETRIE, et Vasant DHAR. « On the Equivalence of Constraint Satisfaction Problems ». Dans *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, 1990. 4

[RvBW06] F. ROSSI, P. van BEEK, et T. WALSH. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006. 1.2

[Rya04] Lawrence RYAN. « *Efficient algorithms for clause-learning SAT solvers* ». PhD thesis, Simon Fraser University, 2004. 2.1.5

[SBB+04a] T. SANG, F. BACCHUS, P. BEAME, H.A. KAUTZ, et T. PITASSI. « Combining Component Caching and Clause Learning for Effective Model Counting ». Dans *Proc. of SAT'04*, 2004. 2

[SBB+04b] Tian SANG, Fahiem BACCHUS, Paul BEAME, Henry A. KAUTZ, et Toniann PITASSI. « Combining Component Caching and Clause Learning for Effective Model Counting ». Dans *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. 2.3, 2.3.1, 4.3.3

[Sch96] R. SCHRAG. « Compilation for critically constrained knowledge bases ». Dans *Proc. of AAAI'96*, pages 510–515, 1996. 4.3.3

[SF94] Daniel SABIN et Eugene C. FREUDER. « Contradicting Conventional Wisdom in Constraint Satisfaction ». Dans *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*, pages 10–20, 1994. 2.2.3

[SG97] Barbara M. SMITH et Stuart A. GRANT. « Modelling Exceptionally Hard Constraint Satisfaction Problems ». Dans Gert SMOLKA, éditeur, *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, volume 1330 de *Lecture Notes in Computer Science*, pages 182–195. Springer, 1997. 6

[Sha40] Claude Elwood SHANNON. « A symbolic analysis of relay and switching circuits ». Thesis (m.s.), Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1940. 2

[SM14] Mirko STOJADINOVIC et Filip MARIC. « meSAT : multiple encodings of CSP to SAT ». *Constraints*, 19(4) :380–403, 2014. 1.3

[SNC09] M. SOOS, K. NOHL, et C. CASTELLUCCIA. « Extending SAT Solvers to Cryptographic Problems ». Dans *Proc. of SAT'09*, pages 244–257, 2009. 2

[SW96] Martin SAUERHOFF et Ingo WEGENER. « On the complexity of minimizing the OBDD size for incompletely specified functions ». *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(11) :1435–1437, 1996. 3.2.2

[TB] Naoyuki TAMURA et Mutsunori BANBARA. « Sugar - a SAT-based constraint solver. ». http://bach.istc.kobe-u.ac.jp/sugar/. 1.3

[Thu06] Marc THURLEY. « sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP ». Dans *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pages 424–429, 2006. 2.3, 2.3.1

[Tod89] Seinosuke TODA. « On the Computational Power of PP and +P ». Dans *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 514–519, 1989. 2.3

[Tse68] G.S. TSEITIN. « *On the complexity of derivation in propositional calculus* », Chapitre Structures in Constructive Mathematics and Mathematical Logic, pages 115–125. Steklov Mathematical Institute, 1968. 1.1.3

[TTB11] T. TANJO, N. TAMURA, et M. BANBARA. « A Compact and Efficient SAT-Encoding of Finite Domain CSP ». Dans *Proc. of SAT'11*, pages 375–376, 2011. 5.4

[TTB12] Tomoya TANJO, Naoyuki TAMURA, et Mutsunori BANBARA. « Azucar : A SAT-Based CSP Solver Using Compact Order Encoding - (Tool Presentation) ». Dans *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pages 456–462, 2012. 1.3, 5.4.1

[TTKB09] Naoyuki TAMURA, Akiko TAGA, Satoshi KITAGAWA, et Mutsunori BANBARA. « Compiling finite linear CSP into SAT ». *Constraints*, 14(2) :254–272, 2009. 1.3, 1.3.1, 5.4.1

[US94] T.E. URIBE et M.E. STICKEL. « Ordered binary decision diagrams and the Davis-Putnam procedure ». Dans *Proc. of the* 1^{st} *International Conference on Constraints in Computational Logic (CCL'94)*, 1994. 2

[Vem92] N.R. VEMPATY. « Solving Constraint Satisfaction Problems Using Finite State Automata ». Dans *Proc. of AAAI'92*, pages 453–458, 1992. 5.4

[vHK06] Willem-Jan van HOEVE et Irit KATRIEL. Global Constraints. Dans *Handbook of Constraint Programming*, pages 169–208. 2006. 1.2.1

[Vor13] A. VORONOV. « *On Formal Methods for Large-Scale Product Configuration* ». Ph.D. thesis, Chalmers University, 2013. 8

[Wal99] Toby WALSH. « Search in a small world ». Dans *Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2*, pages 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. 2.1.5

[Wal00] Toby WALSH. « SAT v CSP ». Dans *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, pages 441–456, 2000. 1.3.1, 5.4

[Wal06] Richard J. WALLACE. « Heuristic Policy Analysis and Efficiency Assessment in Constraint Satisfaction Search ». Dans *18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006), 13-15 November 2006, Arlington, VA, USA*, pages 305–314, 2006. 6

[WES04] Wei WEI, Jordan ERENRICH, et Bart SELMAN. « Towards Efficient Sampling : Exploiting Random Walk Strategies ». Dans *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 670–676, 2004. 2.3.2

[WS05] Wei WEI et Bart SELMAN. « A new approach to model counting ». Dans *Proc. of SAT'05*, pages 324–339, 2005. 2.3.2

[Zan03] Bruno ZANUTTINI. « *Acquisition de connaissances et raisonnement en logique propositionnelle* ». PhD thesis, Université de Caen, Juillet 2003. Chapitre 3. 3.2.4, 10

[ZM02] Lintao ZHANG et Sharad MALIK. « The Quest for Efficient Boolean Satisfiability Solvers ». Dans *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 17–36, 2002. 2.1.2

[ZMMM01] Lintao ZHANG, Conor F. MADIGAN, Matthew W. MOSKEWICZ, et Sharad MALIK. « Efficient Conflict Driven Learning in Boolean Satisfiability Solver ». Dans *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285, 2001. 2.1.4

Résumé

Le comptage de modèles (i.e., des affectations de valeurs aux variables qui satisfont toutes les contraintes) constitue un traitement clé pour divers problèmes d'intelligence artificielle dans lesquels la programmation par contraintes est employée. La compilation de connaissances est l'une des approches possibles permettant de réaliser efficacement une telle requête de comptage. Cependant, dans le cas de contraintes propositionnelles, peu de langages offrent la possibilité de compter les modèles d'une formule en temps polynomial. Dans cette thèse, nous introduisons plusieurs nouveaux langages de compilation de formules propositionnelles, basés sur un connecteur peu utilisé jusqu'ici dans les langages cibles pour la compilation, le ou exclusif. Nous définissons en particulier le langage des arbres de décision affine (ADT) et le langage des arbres de décisions affine étendus ($\texttt{EADT}_\wedge$). Contrairement aux arbres de décision standard, les nœuds de décision d'une formule ADT ou $\texttt{EADT}_\wedge$ ne sont pas étiquetés par des variables mais, de façon plus générale, par des clauses affines. Ces langages sont analysés suivant les critères considérés dans la carte de compilation des langages propositionnels afin de pouvoir être comparés aux différents langages déjà inclus dans celle-ci et dans l'objectif d'étendre cette carte de compilation. Nous proposons également un compilateur CNF vers $\texttt{EADT}_\wedge$ que nous avons évalué et nous présentons les résultats expérimentaux obtenus. Ces résultats montrent qu'une approche basée sur le langage $\texttt{EADT}_\wedge$ concurrence, et est parfois même plus performante, que des algorithmes directs (sans compilation) de l'état de l'art pour le comptage de modèles. Nous nous sommes également intéressés à la compilation de connaissances représentées primitivement dans un autre formalisme, à savoir celui des réseaux de contraintes. En effet, il existe jusqu'ici peu de méthodes permettant de compter le nombre de modèles d'un réseau de contraintes. Pour cela, nous avons introduit et étudié le langage cible des graphes de décision décomposables multi-valués (MDDG). Ce langage peut être vu comme une généralisation au cas des variables possiblement non booléennes du langage cible Decision−DNNF pour la logique propositionnelle. MDDG offre les mêmes requêtes et transformations que celles satisfaites par le langage Decision−DNNF, ce qui le rend très attractif. Nous avons développé un compilateur de réseaux de contraintes vers MDDG dans lequel plusieurs ingrédients originaux ont été testés, comme l'utilisation d'une heuristique de choix de variables basée sur la centralité d'intermédiarité ou encore la détection et la prise en compte des contraintes universelles. Les expérimentations ont montré que l'approche par compilation en MDDG est souvent très compétitive. En particulier, le compilateur développé parvient à compiler des réseaux de contraintes qui se trouvent généralement hors de portée des approches basées sur la traduction du réseau en CNF, compilée ensuite en Decision−DNNF. Les expérimentations réalisées ont également montré que l'approche par compilation en MDDG d'obtenir des formes compilées dont la taille se révèle souvent beaucoup plus concise (et ce parfois de plusieurs ordres de grandeur) que celles obtenues par traduction en CNF puis compilation en Decision−DNNF.

Mots-clés: compilation de connaissances, comptage de modèles, logique propositionnelle, réseaux de contraintes