Arbres de décision affine pour le comptage de modèles

Frédéric Koriche Jean-Marie Lagniez Pierre Marquis Samuel Thomas

CRIL-CNRS, Université d'Artois, Lens, France
{koriche,lagniez,marquis,thomas}@cril.univ-artois.fr

Résumé

Le comptage de modèles est une question clé pour un grand nombre de problèmes en intelligence artificielle. Cependant, peu de langages propositionnels offrent la possibilité de compter les modèles efficacement. Dans le but de combler ce manque, nous introduisons le langage des arbres de décision affine étendus (EADT). Un arbre de décision affine étendu est un arbre composé de nœuds de décision affine, et de nœuds de conjonction ou de disjonction affine décomposable ou déterministes. Contrairement aux arbres de décision standard, les nœuds de décision d'une formule EADT ne sont pas étiquetés par des variables mais par des clauses affines. Dans ce papier, nous étudions EADT et plusieurs sous-ensembles de ce langage suivant les critères considérés dans la carte de compilation. Puis, nous décrivons un compilateur CNF-to-EADT et présentons quelques résultats expérimentaux. Ces résultats montrent que la compilation basée sur l'approche EADT concurrence (et est parfois plus performante que) le compteur de modèles Cachet ainsi que l'approche de comptage de modèles par compilation s'appuyant sur le langage d-DNNF.

1 Introduction

Le comptage de modèles est une question clé pour un grand nombre de problèmes en Intelligence Artificielle, comme l'inférence dans les réseaux bayésiens [19, 1, 28, 10], les problèmes de configuration [14], de planification et d'ordonnancement [17, 29, 11]. Cependant, ce problème est difficile (#P-complet) [35]. De ce fait, peu de langages propositionnels offrent le comptage *exact* de modèles de manière efficace [26].

La carte de compilation (KC), présentée par Darwiche et Marquis [9], enrichie par plusieurs auteurs [37, 34, 21, 13, 8, 20, 2] est une évaluation multi-critère de langages propositionnels. Les langages y sont comparés selon les requêtes et transformations qu'ils supportent en temps polynomial, ainsi que leur efficacité spatiale (c'est-à-dire leur capacité à représenter les informations en utilisant plus ou moins d'espace).

Parmi les langages étudiés et classés dans la carte de compilation, seul le langage d-DNNF [7] et ses sous-ensembles $\text{OBDD}_<$ [3], FBDD [15] et SDD [8] satisfont la requête **CT** (le comptage de modèles).

Un autre langage intéressant supportant **CT** est l'ensemble des formules affines AFF [30], où une formule affine est une conjonction finie de clauses affines (alias XOR-clauses). Cependant, AFF n'est pas un langage *complet*. En effet, certaines formules propositionnelles ne peuvent pas être représentées par des conjonctions de formules affines. C'est le cas par exemple de la clause $x_1 \vee x_2$

En associant les idées des formules affines et des arbres de décision, ce papier introduit une nouvelle famille de langages propositionnels complets et satisfaisant **CT**. Cette famille de langages est basée sur la classe des *arbres de décision affine étendus* (EADT). Un arbre de décision affine étendu est un arbre composé de nœuds de décision affine et de nœuds de conjonction ou de disjonction affine décomposable. Contrairement aux arbres de décision habituels, les nœuds de décisions d'une formule EADT sont étiquetés par des clauses affines au lieu de variables. Notre étude porte sur plusieurs sous-ensembles de EADT, incluant ADT (l'ensemble des arbres de décision affine dans lesquels les nœuds de conjonction et de disjonction sont exclus), EDT (l'ensemble des arbres de décision étendus dans lesquels les nœuds de conjonction et de disjonction sont autorisés), et l'intersection DT de ADT et EDT.

Nous prouvons que ADT et son sous-ensemble DT satisfont les mêmes requêtes et transformations qu'offrent les diagrammes de décision binaire ordonnés ($\text{OBDD}_<$). Nous montrons aussi que EADT et son sous-ensemble EDT satisfont les mêmes requêtes que celles offertes par les d-DNNF et offrent plus de transformations ($\neg\mathbf{C}$). Nous montrons aussi que $\text{OBDD}_<$, CNF et

DNF ne sont pas au moins aussi succincts que ADT ou EADT, et que EADT est strictement plus succinct que ADT.

Enfin, nous décrivons un compilateur CNF-to-EADT (qui peut être simplifié en un compilateur ciblant les langages ADT, EDT ou DT). Nous avons appliqué ce compilateur à plusieurs jeux d'essai issus de différents domaines. Cette évaluation empirique vise à répondre à deux questions : (1) Que vaut une approche basée sur le comptage de modèles par la compilation en EADT comparée à une approche de comptage de modèles directe avec un compteur de modèles de l'état de l'art ? (2) Que vaut une approche de compilation en EADT comparée à une méthode de compilation en d-DNNF ? Nos résultats expérimentaux montrent qu'une approche de compilation basée sur les EADT concurrence chacune des méthodes et se montre, parfois, plus performante.

Après l'introduction de quelques notions en section 2, nous définissons EADT ainsi que ses sous-ensembles et nous les étudions en section 3. À la section 4, nous décrivons notre compilateur, puis à la sections 5, nous présentons et discutons quelques résultats expérimentaux. Enfin, nous concluons ce papier en section 6. [1]

2 Préliminaires

Dans cet article, nous supposons que le lecteur est familier avec la logique propositionnelle (en particulier les notions de modèle, de cohérence, de validité, d'implication et d'équivalence). Tous les langages présentés sont définis sur un ensemble fini PS de variables booléennes, plus les constantes booléennes $\top$ (vrai) et $\bot$ (faux).

Formule affine. Un littéral (sur PS) est un élément $x \in PS$ (un littéral positif) ou sa négation $\neg x$ (un littéral négatif), ou $\top$ (vrai) ou $\bot$ (faux). Une clause affine (alias XOR-clause) est une XOR-disjonction finie de littéraux (le connecteur XOR est noté $\oplus$). Elle est unaire lorsqu'elle contient exactement un littéral. Toute clause affine δ peut être réécrite en temps linéaire en une clause simplifiée, c'est-à-dire en une XOR-disjonction finie de littéraux positifs ayant chacun une seule occurrence dans la clause, plus éventuellement une occurrence de $\top$ (on utilise les équivalences $\neg x \equiv x \oplus \top$, $x \oplus x \equiv \bot$ et $x \oplus \bot \equiv x$). Par exemple, la clause affine $\neg x \oplus x \oplus \neg y \oplus \neg z$ peut être transformée en temps linéaire en la clause affine simplifiée équivalente $y \oplus z \oplus \top$. $var(\delta)$ dénote l'ensemble de variables apparaissant dans δ. Une formule affine est une conjonction finie de clauses affines.

1. Les preuves des résultats sont dans la version anglaise de l'article disponible à l'adresse : http://cril.univ-artois.fr/ADT

Compilation. Pour des raisons de place, nous supposons que le lecteur est familier avec les langages CNF, DNF, $\text{OBDD}_<$, SDD, BDD, FBDD, d-DNNF_T, d-DNNF et DAG-NNF qui sont utilisés par la suite (voir [9, 25, 8] pour les définitions formelles). Les requêtes basiques considérées dans la carte de compilation incluent celles de cohérence **CO**, de validité **VA**, d'implication clausale **CE**, des implicants **IM**, d'équivalence **EQ**, d'implication **SE**, du comptage de modèles **CT**, de l'énumération de modèles **ME**. Les transformations basiques prises en compte sont le conditionnement **CD**, les clôtures (possiblement bornées) par les connecteurs ($\wedge$**C**, $\wedge$**BC**, $\vee$**C**, $\vee$**BC**, $\neg$**C**), et l'oubli (**FO**, **SFO**). Formellement :

Définition 1 *Soit $\mathcal{L}$ un langage propositionnel*

- *Cohérence (***CO***)*
  $\mathcal{L}$ vérifie la propriété de cohérence si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toute formule Σ de $\mathcal{L}$ vrai *si Σ est cohérente et* faux *sinon.*
- *Validité (***VA***)*
  $\mathcal{L}$ vérifie la propriété de validité si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toute formule Σ de $\mathcal{L}$ vrai *si Σ est valide et* faux *sinon.*
- *Implication clausale (***CE***)*
  $\mathcal{L}$ vérifie la propriété d'implication clausale si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toute formule Σ de $\mathcal{L}$ et toute clause γ vrai *si $\Sigma \models \gamma$ et* faux *sinon.*
- *Équivalence (***EQ***)*
  $\mathcal{L}$ vérifie la propriété d'équivalence si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toutes formules Σ et ϕ de $\mathcal{L}$, vrai *si $\Sigma \equiv \phi$ et* faux *sinon.*
- *Implication (***SE***)*
  $\mathcal{L}$ vérifie la propriété d'implication générale si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toutes formules Σ et ϕ de $\mathcal{L}$, vrai *si $\Sigma \models \phi$ et* faux *sinon.*
- *Test d'implicant (***IM***)*
  $\mathcal{L}$ vérifie la propriété du test d'implicant si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toute formule Σ de $\mathcal{L}$ et tout terme γ, vrai *si $\gamma \models \Sigma$ et* faux *sinon.*
- *Comptage de modèles (***CT***)*
  $\mathcal{L}$ satisfait la propriété de comptage de modèles si et seulement s'il existe un algorithme en temps polynomial qui retourne pour toute formule Σ de $\mathcal{L}$ un entier naturel qui est le nombre de modèles de Σ.
- *Énumération de modèles (***ME***)*
  $\mathcal{L}$ satisfait la propriété d'énumération de modèles 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 est son nombre de modèles (sur les variables de Σ).

Définition 2 *Soit $\mathcal{L}$ un langage propositionnel*

— *Conditionnement (**CD**)*
$\mathcal{L}$ satisfait la propriété de conditionnement si et seulement s'il existe un algorithme en temps polynomial qui pour toute formule Σ de $\mathcal{L}$ et tout terme γ retourne une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma|\gamma$.
— *Oubli (**FO**)*
$\mathcal{L}$ satisfait la propriété d'oubli si et seulement s'il existe un algorithme en temps polynomial qui pour chaque formule Σ de $\mathcal{L}$ et chaque sous-ensemble $\mathcal{X}$ de variables de PS retourne une formule de $\mathcal{L}$ logiquement équivalente à $\exists\mathcal{X}.\Sigma$
— *Oubli singleton (**SFO**)*
*Si la propriété **FO** s'applique pour un singleton $\mathcal{X}$, on dit que $\mathcal{L}$ satisfait **SFO**.*
— *Négation ($\neg$**C**)*
$\mathcal{L}$ satisfait la propriété de négation si et seulement s'il existe un algorithme en temps polynomial qui permet de transformer chaque formule Σ de $\mathcal{L}$ en une formule de $\mathcal{L}$ logiquement équivalente à $\neg\Sigma$.
— *Conjonction ($\wedge$**C**), disjonction ($\vee$**C**)*
$\mathcal{L}$ satisfait la propriété de conjonction (respectivement disjonction) si et seulement s'il existe un algorithme en temps polynomial qui permet de transformer chaque ensemble fini de formules $\Sigma_1, \ldots, \Sigma_n$ de $\mathcal{L}$ en une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma_1 \wedge \ldots \wedge \Sigma_n$ (respectivement $\Sigma_1 \vee \ldots \vee \Sigma_n$).
— *La conjonction bornée ($\wedge$**BC**), disjonction bornée ($\vee$**BC**)*
$\mathcal{L}$ satisfait la propriété de conjonction bornée (respectivement disjonction bornée) si et seulement s'il existe un algorithme en temps polynomial qui permet de transformer chaque paire de formules Σ et Φ de $\mathcal{L}$ en une formule de $\mathcal{L}$ logiquement équivalente à $\Sigma \wedge \Phi$ (respectivement $\Sigma \vee \Phi$).

Définition 3 *Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages propositionnels.*

— *$\mathcal{L}_1$ est au moins aussi succinct que $\mathcal{L}_2$, noté $\mathcal{L}_1 \leq_s \mathcal{L}_2$, si et seulement s'il existe un polynôme p tel que pour toute formule $\phi \in \mathcal{L}_2$, il existe une formule équivalente $\psi \in \mathcal{L}_1$ où $|\psi| \leq p(|\phi|)$;*
— *$\mathcal{L}_1$ est polynomialement traduisible en $\mathcal{L}_2$, si et seulement s'il existe un algorithme en temps polynomial f tel que pour tout $\phi \in \mathcal{L}_1$, $f(\phi) \in \mathcal{L}_2$ et $f(\phi) \equiv \phi$.*

$<_s$ est la partie asymétrique de $\leq_s$, c'est-à-dire que $\mathcal{L}_1 <_s \mathcal{L}_2$ si et seulement si $\mathcal{L}_1 \leq_s \mathcal{L}_2$ et $\mathcal{L}_2 \not\leq_s \mathcal{L}_1$. Quand $\mathcal{L}_1$ est polynomialement traduisible en $\mathcal{L}_2$, toute requête en temps polynomial dans $\mathcal{L}_2$ peut aussi être satisfaite en temps polynomial dans $\mathcal{L}_1$. De même, toute requête non supportée par $\mathcal{L}_1$ en temps polynomial sauf si P=NP ne peut pas être supportée par $\mathcal{L}_2$ sauf si P=NP.

3 La famille affine

Les langages propositionnels considérés dans notre famille sont des sous-ensembles du langage général des réseaux de décision affine (ADN), défini comme suit.

Définition 4 *Un réseau de décision affine (ADN) est 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 $\wedge$ ou $\vee$ (avec un nombre arbitraire de fils) ou des nœuds de décision affine, c'est-à-dire des nœuds binaires 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.*

La taille $|\Delta|$ d'une formule de type (ADN) Δ est la somme de son nombre d'arcs et de la taille cumulée des clauses affines utilisées pour étiqueter les sommets. Pour tout nœud N d'une formule ADN Δ, $Var(N)$ 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 $Var(N) = \cup_{i=1}^{k} Var(N_i)$.

On définit aussi Δ_N comme la formule ADN enraciné en N.

Clairement $Var(\Delta) = Var(R_\Delta)$ (avec R_Δ la racine de Δ) peut être calculé en temps linéaire en la taille de Δ. Toute formule ADN Δ est interprétée comme une formule propositionnelle $I(\Delta)$ sur $Var(\Delta)$, avec $I(\Delta) = I(R_\Delta)$ défini inductivement 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 $\bigvee_{i=1}^{k} I(N_i)$).

$\|\Delta\|$ désigne enfin le nombre de modèles de Δ sur $Var(\Delta)$.

Le langage DAG-NNF considéré dans [9] est polynomialement traduisible en un sous-ensemble de ADN, où

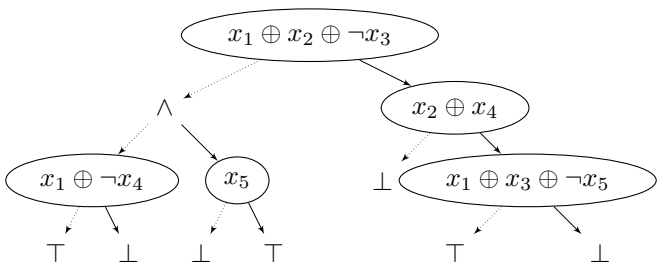

FIGURE 1 – Une formule EADT. Les arcs en pointillés (resp. pleins) représentent le lien entre N et N_- (resp. N_+). La formule ayant pour racine le nœud étiqueté $x_2 \oplus x_4$ est une formule ADT.

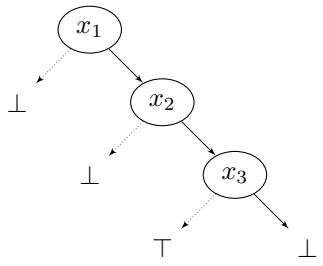

FIGURE 2 – Un terme compilé en formule ADT

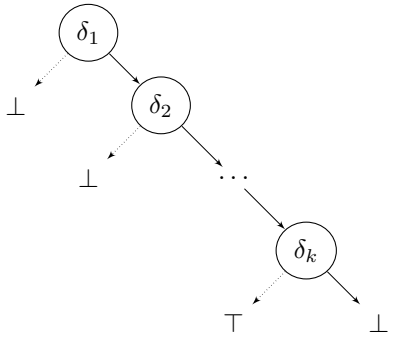

FIGURE 3 – Une formule affine compilée en formule ADT

les nœuds de décision affine possèdent uniquement des feuilles comme fils. En effet, chaque nœud étiqueté par un littéral positif x (respectivement négatif $\neg x$) d'un DAG-NNF est équivalent à un nœud de décision affine N étiqueté par x, tel que $N_- = \bot$ (respectivement $= \top$) et $N_+ = \top$ (respectivement $= \bot$).

ADN est un langage de représentation très succinct mais offrant peu de requêtes en temps polynomial. En particulier, il ne satisfait pas le comptage de modèles **CT** sauf si P = NP. C'est pourquoi nous nous concentrons sur des sous-ensembles propres de ADN.

Nous commençons avec le langage EADT, une classe de formules structurées en arbre, à base de nœuds de décision affine et de nœuds $\wedge$, $\vee$ affine décomposables. Un nœud $\wedge$ (respectivement $\vee$) N ayant pour fils $N_1, \ldots, N_k$ dans une formule ADN Δ est dit 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) pour tout nœud de décision affine $N' = \langle \delta, N_-, N_+ \rangle$ de Δ qui est un ancêtre de N, au plus un fils N_i de N est tel que $Var(N_i) \cap var(\delta) \neq \varnothing$.

Si nous ne prenons en compte que la première condition, alors le nœud N est dit (classiquement) décomposable.

Définition 5 *Un arbre de décision affine étendu* (EADT) *est un arbre fini dont les feuilles sont étiquetées par $\top$ ou $\bot$ et les nœuds internes sont des nœuds de décision affine oudes nœuds $\wedge$ ou $\vee$ affine décomposables.*

Un exemple de formule EADT est présenté à la figure 1. Plusieurs sous-classes intéressantes de EADT peuvent être définies :

Définition 6

— *L'ensemble des arbres de décision affine* ADT *est le sous-ensemble de* EADT *des arbres finis dont les feuilles sont $\top$ ou $\bot$ et les nœuds internes sont des nœuds de décision affine*
— *L'ensemble des arbres de décision étendus* EDT*est le sous-ensemble de* EADT *des arbres dont les nœuds de décision sont étiquetés par des clauses affines unaires*
— *L'ensemble des arbres de décision* DT *est l'intersection de* ADT *et* EDT.

Nous pouvons facilement prouver que le langage des termes TE et celui des clauses CL sont polynomialement traduisibles en DT, donc en ADT, EDT et EADT. Par exemple, le terme $x_1 \wedge x_2 \wedge \neg x_3$ est équivalent à la formule DT de la figure 2.

Le langage AFF est polynomialement traduisible en ADT (et donc en son sur-ensemble EADT). En effet, une formule affine $\Delta = \bigwedge_{i=1}^{k} \delta_i$ où chaque δ_i $(i \in 1, \ldots, k)$ est une clause affine, est équivalent à la formule ADT de la figure 3 qui peut être calculée à partir de Δ en temps linéaire.

Contrairement à TE, CL et AFF, la classe DT et ses sur-ensembles ADT, EDT et EADT 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 est un sous-ensemble de ADT. La propriété de complétude est aussi satisfaite par $\text{ODT}_<$, le sous-ensemble de DT des formules Δ où chaque chemin de la racine aux feuilles

de Δ respecte un ordre strict et total $<$ (les variables qui étiquetent les nœuds de décision sont ordonnées sur le chemin suivant $<$). Il apparait clairement que $\texttt{ODT}_<$ est également un sous-ensemble de $\texttt{OBDD}_<$ (plus précisément, $\texttt{ODT}_<$ est l'intesection de DT et $\texttt{OBDD}_<$), et CL et TE sont polynomialement traduisibles en $\texttt{OBDD}_<$.

Nous montrons maintenant comment toute formule EADT Δ peut être transformée en temps linéaire en un arbre $T(\Delta)$ où les nœuds internes sont des nœuds $\wedge$ ou $\vee$ affine décomposables et les feuilles sont étiquetées par des formules affines. La transformation T consiste à réécrire Δ en la parcourant de façon descendante et retenant les ensembles de clauses affines (ces ensembles sont les valeurs d'un attribut hérité a défini pour chaque nœud N de Δ) sur les chemins de Δ pendant la transformation. T est défini de manière récursive de la façon suivante avec $N = R_\Delta$ et $a(R_\Delta) = \varnothing$ au début, puis :

- si $N = \langle \delta, N_-, N_+ \rangle$, alors $T(N) = T(N_-) \vee T(N_+)$, $a(N_-) = a(N) \cup \{\delta \oplus \top\}$, et $a(N_+) = a(N) \cup \{\delta\}$;
- si $N = \bigwedge_{i=1}^k N_i$, alors $T(N) = \bigwedge_{i=1}^k T(N_i)$, et pour tout $i \in 1, \ldots, k$, $a(N_i) = \{\delta \in a(N) \mid var(\delta) \cap Var(N_i) \neq \varnothing\}$;
- si $N = \bigvee_{i=1}^k N_i$, alors $T(N) = \bigvee_{i=1}^k T(N_i)$, et pour tout $i \in 1, \ldots, k$, $a(N_i) = \{\delta \in a(N) \mid var(\delta) \cap Var(N_i) \neq \varnothing\}$;
- si $N = \top$, alors $T(N) = \bigwedge_{\delta \in a(N)} \delta$;
- si $N = \bot$, alors $T(N) = \bot$.

Par construction, la transformation T consiste à remplacer chaque nœud de décision affine par un nœud déterministe $\vee$, chaque nœud affine décomposable $\wedge$ (resp. $\vee$) par un nœud décomposable (classique) $\wedge$ (resp. $\vee$). Ainsi, quand Δ est une formule ADT, $T(\Delta)$ est simplement une disjonction déterministe de formules affines, et quand Δ est une formule DT, $T(\Delta)$ est simplement une formule DNF déterministe.

En s'appuyant sur cette transformation, il est facile de montrer que EADT satisfait la requête **CT**. La preuve est par induction structurelle sur $\phi = T(\Delta)$. Premièrement, $\|\phi\|$ peut être calculé en temps polynomial quand ϕ est une formule affine. En effet, ϕ peut être mise en temps polynomial sous forme triangulée ϕ^r, et quand $Var(\phi^r)$ contient n variables et ϕ^r contient k clauses affines, $\|\phi\|$ est égal à 2^{n-k}. Cela résout le cas de base. Ensuite, par induction :

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

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

- si $\phi = \bigvee_{i=1}^k \phi_i$ (où $\bigvee$ est un nœud $\vee$ décomposable),

$$\|\phi\| = 2^{|\,Var(\phi)|} - \prod_{i=1}^k (2^{|\,Var(\phi_i)|} - \|\phi_i\|)$$

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

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

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

TABLE 1 – Requêtes. √ signifie "satisfait" et ○ signifie "ne satisfait pas sauf si P = NP."

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

TABLE 2 – Transformations. √ signifie "satisfait" et ○ signifie "ne satisfait pas sauf si P = NP".

Nos résultats concernant les requêtes et transformations de la carte de compilation sont résumés à la proposition 1. Les classes d-DNNF et $\texttt{OBDD}_<$ (qui ne sont pas des sous-ensembles de EADT) sont présentés à titre de comparaison.

Proposition 1 *Les résultats présentés dans les tableaux 1 et 2 sont corrects.*

En résumé, ADT et sa sous-classe DT sont équivalents à $\texttt{OBDD}_<$ au regard des requêtes et transformations qu'ils satisfont. De même, EADT et sa sous-classe EDT sont essentiellement équivalent à d-DNNF pour les requêtes et transformations qu'ils satisfont. En particulier, EDT, EADT et d-DNNF ne satisfont pas **SE** sauf si P = NP, et nous ne savons pas s'ils satisfont **EQ**. Nous ne savons pas non plus si d-DNNF satisfait **¬C**, mais cette transformation peut être effectuée en temps linéaire pour EADT et pour EDT.

Le graphe d'inclusion des différents langages considérés est donné à la figure 4. $\texttt{d-DNNF}_\texttt{T}$ est la classe introduite par Pipatsrisawat and Darwiche [25], l'ensemble des formules d-DNNF qui respectent un vtree T. Étant donné ce graphe d'inclusion et du fait que DT (respectivement EDT) ne satisfait pas plus de requêtes

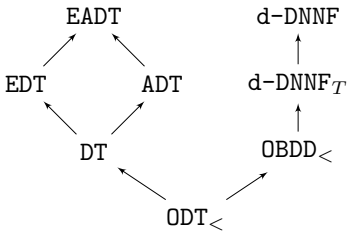

FIGURE 4 – Graphe d'inclusion. $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ indique que $\mathcal{L}_1 \subseteq \mathcal{L}_2$.

ou de transformations que ADT (respectivement EADT), nous pouvons déduire que DT (respectivement EDT) ne peut pas être un meilleur choix que ADT (respectivement EADT) du point de vue de la carte de compilation. C'est pourquoi nous nous concentrons sur les langages ADT et EADT. Nous avons identifié les propriétés d'efficacité spatiale suivantes pour ces langages :

Proposition 2

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

Sur la base de ces résultats, on peut conclure que OBDD$_<$ ne domine pas ADT du point de vue de la carte de compilation. De plus, comme ADT $\subseteq$ EADT, OBDD$_<$ ne domine pas EADT. Nos résultats de concision montrent qu'aucun des langages « plats » CNF et DNF n'est au moins aussi succinct que ADT ou EADT. Bien que nous ne connaissions pas comment d-DNNF et EADT se comparent du point de vue de l'efficacité spatiale, nous pouvons néanmoins conclure que la sous-classe d-DNNF$_T$ de d-NNF ne domine ni ADT, ni EADT.

4 Un compilateur CNF-to-EADT

Pour compiler une formule propositionnelle quelconque en un arbre de décision affine étendu, on s'appuie sur une forme généralisée de l'expansion de Shannon. Soient deux formules Δ et δ, et une variable x, nous notons $\Delta\,|_{x\leftarrow\delta}$ la formule obtenue en replacant toute occurrence de x dans Δ par δ. Avec cette notation, nous généralisons l'expansion de Shannon ; pour toute formule propositionnelle Δ et δ et toute variable x, nous avons :

$$\Delta \equiv ((x \Leftrightarrow \neg\delta) \wedge \Delta\,|_{x\leftarrow\neg\delta}) \vee ((x \Leftrightarrow \delta) \wedge \Delta\,|_{x\leftarrow\delta})$$

On peut observer que l'on retrouve l'expansion de Shannon standard [32] en considérant $\delta = \top$. La validité de l'expansion de Shannon généralisée vient du fait que Δ est équivalent à $((x \Leftrightarrow \neg\delta) \wedge \Delta) \vee ((x \Leftrightarrow \delta) \wedge \Delta)$, et du fait que pour tout formule propositionnelle δ (ou sa négation), l'expression $(x \Leftrightarrow \delta) \wedge \Delta$ est équivalente à $(x \Leftrightarrow \delta) \wedge \Delta\,|_{x\leftarrow\delta}$.

Dans notre cas, Δ est une formule ECNF et δ est une clause affine. Le langage des ECNF est l'ensemble de toutes les conjonctions finies de clauses étendues, où une clause étendue est une disjonction finie de clauses affines. Par exemple, $x_1 \vee (x_2 \oplus x_3 \oplus \top) \vee (x_1 \oplus x_3)$ est une clause étendue. La conversion d'une CNF en ECNF s'effectue en temps linéaire de manière évidente. Comme $x \Leftrightarrow \neg\delta$ est équivalent à $x \oplus \delta$, et comme $x \Leftrightarrow \delta$ est équivalent à $x \oplus \delta \oplus \top$, l'expansion généralisée de Shannon peut être vue comme suivant la règle de branchement suivante :

$$\Delta \equiv ((x \oplus \delta) \wedge \Delta\,|_{x\leftarrow\delta\oplus\top}) \vee ((x \oplus \delta \oplus \top) \wedge \Delta\,|_{x\leftarrow\delta})$$

L'algorithme de compilation. L'algorithme 1 décrit en pseudo-code notre compilateur eadt se basant sur l'expansion généralisée de Shannon. Il prend en entrée une formule ECNF Δ et retourne une formule EADT logiquement équivalente à Δ. Les deux premières lignes gèrent les deux cas particuliers où Δ est valide ou contradictoire. Dans les deux cas, la feuille correspondante est retournée.

Nous notons au passage que le problème d'insatisfiabilité (respectivement de validité) pour EADT 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 d'insatisfiabilité. 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, ce qui peut être testé en temps polynomial.

À la ligne 3, Δ est vue comme une conjonction décomposable de composantes $\Delta_1, \cdots, \Delta_k$. Ces composantes sont compilées de manière récursive en formules EADT puis jointes par un nœud $\wedge$ en utilisant la fonction aNode (ligne 4). Notons que la décomposition s'effectue avant le branchement. Lorsque Δ ne possède plus qu'une composante, le compilateur choisit une clause affine $x \oplus \delta$ dans laquelle toute variable possède au moins une occurrence dans Δ (ligne 5). Puis nous effectuons le branchement sur cette clause en utilisant l'expansion généralisée de Shannon (ligne 6). Enfin, la fonction dNode retourne un nœud de décision correspondant. Privé des lignes 3 et 4, le compilateur CNF-to-EADT se réduit à un compilateur CNF-to-ADT.

La terminaison de l'algorithme 1 est garantie par le fait que par définition, une clause affine simplifiée δ dans $x \oplus \delta$ est une clause affine qui ne contient pas x.

Algorithme 1 : eadt(Δ)

entrée : une formule ECNF Δ
sortie : une formule EADT équivalente à Δ

1 si $\Delta \equiv \top$ alors return leaf($\top$)

2 si $\Delta \equiv \bot$ alors return leaf($\bot$)

3 soient $\Delta_1, \cdots, \Delta_k$ les composantes connexes de Δ
4 si $k>1$ alors return aNode(eadt(Δ_1), $\cdots$, eadt(Δ_k))

5 choisir une clause affine simplifiée $x \oplus \delta$ telle que

$$var(x \oplus \delta) \subseteq Var(\Delta)$$

6 return dNode($x \oplus \delta$, eadt($\Delta \mid_{x \leftarrow \delta \oplus \top}$), eadt($\Delta \mid_{x \leftarrow \delta}$))

Comme ni $\Delta \mid_{x \leftarrow \delta \oplus \top}$ ni $\Delta \mid_{x \leftarrow \delta}$ ne contiennent x, les étapes 5 et 6 ne peuvent être appliquées qu'un nombre fini de fois. De plus, la formule EADT retournée par l'algorithme est assurée de satisfaire la règle de décomposition affine. Cette propriété peut être prouvée par induction sur la structure de l'arbre résultat : l'unique cas non trivial est lorsque l'arbre contient un nœud $\wedge$ avec pour parents N et pour fils $N_1, \cdots, N_k$, où chaque N_i est créé en appelant eadt sur la composante connexe Δ_{N_i} de la formule Δ. Comme chaque clause parent N est de la forme $x \oplus \delta$, où x est exclu de δ, et comme les composantes connexes distinctes ne partagent pas de variables, $x \oplus \delta$ n'intersecte qu'au plus une des composants $\Delta_1, \cdots, \Delta_k$. Comme l'expansion généralisée de Shannon est valide, l'algorithme eadt est correct.

Implémentation. L'algorithme 1 a été implémenté sur la base du solveur état de l'art MiniSat [12], qui s'appuie sur une architecture CDCL (*Conflict Driven Clause Learning*) [22]. Nous avons étendu MiniSat aux formules ECNF. 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*) [22]. Plus précisément, pour chaque clause étendue C de Δ, le score de C 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. Nous sélectionnons une clause étendue C de Δ de score maximal, et les variables de C sont triées par ordre décroissant selon leur score VSIDS. La variable x sélectionnée est la première variable de la liste ainsi créée, et la clause affine δ est formée des $k-1$ autres variables de la liste. Notons que sélectionner toutes les variables de la même clause étendue C de Δ permet d'éviter de générer des connexions entre les variables qui ne sont pas déjà connectées dans le graphe de contraintes de Δ. Nous tirons aussi 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. Pour nos expérimentations, nous avons borné la taille des clauses affines à $k = 2$ et utilisé la méthode de filtrage jusqu'à une profondeur 5.

5 Expérimentations

Protocole. Le protocole empirique que nous avons suivi est très proche de celui appliqué dans [31] (et d'autres papiers) où le but était de déterminer quand une approche d'implication clausale basée sur la compilation peut se montrer plus satisfaisante qu'une approche directe, non compilée.

Nous avons considéré plusieurs jeux d'essais constitués de formules CNF issus de différents domaines de la SAT LIBrary [2]. Nous avons uniquement sélectionné les jeux d'essais ayant plus d'une solution. Pour chaque instance CNF Δ, 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 Δ, suivant une distribution uniforme. Le signe de chaque littéral est aussi choisi avec une probabilité $\frac{1}{2}$. L'objectif est de compter le nombre de modèles de la formule conditionnée $\Delta \mid \gamma$ 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 requêtes 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 [3] [27]. 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,

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

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

2. SATLIB est disponible sur le site www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html

3. Cachet est disponible à www.cs.rochester.edu/~kautz/Cachet/index.htm

Instance			Cachet		c2d				eadt			
name	$\#var$	$\#cla$	$\#F$	$\overline{Q}$	C	$\overline{Q}$	α	β	C	$\overline{Q}$	α	β
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**
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	$+\infty$	-	-	-	-
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**
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	$+\infty$
ssa7552-038	1501	3575	1000	0.634	20.99	0.042	0.065	**35**	-	-	-	-

TABLE 3 – Quelques résultats expérimentaux

— deux approches par compilation d-DNNF et EADT ont été sélectionnées. Nous avons utilisé le compilateur c2d [4] afin de compiler en d-DNNF [5] et notre propre compilateur pour compiler en EADT. Pour chaque langage $\mathcal{L}$ parmi d-DNNF et EADT, Δ a tout d'abord été transformée en sa forme compilée $\Delta^* \in \mathcal{L}$. Le temps de compilation $C_{\mathcal{L}}$ nécessaire pour compiler Δ^* et le temps de réponse moyen aux requêtes $\overline{Q}_{\mathcal{L}}$ ont été mesurés. Pour chaque approche, nous avons aussi calculé deux ratios :

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

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 $\alpha_{\mathcal{L}} \geq 1$.

Résultats. Le tableau 3 présente les résultats obtenus. Chaque ligne correspond à une instance CNF Δ identifiée par la colonne la plus à gauche. Les deux premières colonnes donnent respectivement le nombre $\#var$ de variables de Δ et le nombre $\#cla$ de clauses de Δ, les autres colonnes indiquent les valeurs mesurées. Les différents temps de calculs rapportés sont 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 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 (non donné dans le tableau par manque d'espace) 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 d-DNNF et une pour EADT).

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

Finalement, nos expérimentations montrent que la compilation en EADT 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 EADT et d-DNNF sont du même ordre de grandeur, mais la formule EADT est légèrement plus efficace pour traiter les requêtes, grâce à sa structure arborescente.

4. c2d est disponible à reasoning.cs.ucla.edu/c2d/

5. Nous avions aussi prévu d'utiliser le compilateur d-DNNF Dsharp [23] mais malheureusement, nous avons rencontré le même problème celui que mentionné dans [36] pour le lancer, ce qui nous a empêchés de l'utiliser.

6 Conclusion

Le langage propositionnel EADT introduit dans cet article apparaît comme un langage de compilation très attractif quand **CT** est une requête clé. En particulier, EADT offre les mêmes requêtes que d-DNNF et offre plus de transformations. Le sous-ensemble ADT de EADT satisfait toutes les requêtes et les mêmes transformations que celles offertes par le langage $\texttt{OBDD}_<$. De plus, $\texttt{OBDD}_<$ n'est pas au moins aussi succinct que ADT, ce qui place ADT comme un challenger possible des $\texttt{OBDD}_<$. En pratique, l'approche par compilation en EADT pour compter les modèles apparaît compétitive avec le compteur de modèles Cachet et l'approche par compilation en d-DNNF.

Ce travail ouvre plusieurs perspectives pour de futures recherches. D'un point de vue théorique, une extension naturelle de ADT est l'ensemble des DAG finis ne possédant qu'un nœud racine, où les feuilles sont étiquetées par $\top$ ou $\bot$, et les nœuds internes sont des nœuds de décision affine. Cependant, ce langage n'est pas un langage intéressant du point de vue de la compilation, car il contient le langage des diagrammes de décision binaire (BDD) [3] comme sous-ensemble, et BDD ne satisfait *aucune* requête proposée dans la carte de compilation [9], sauf si $\mathsf{P} = \mathsf{NP}$. Ainsi, le problème de déterminer des classes intéressantes de graphes de décision affine offrant le comptage de modèles semble un problème intéressant à creuser.

D'un point de vue pratique, différentes pistes pour améliorer notre compilateur existent. Il serait intéressant de profiter des techniques de pré-traitement développée pour SAT [24, 16] afin de simplifier la formule CNF en entrée avant de la compiler. De plus, il pourrait être utile d'utiliser l'élimination de Gauss pour traiter plus efficacement (voir par exemple [18, 5, 33]) les instances contenant des sous-problèmes correspondant à des clauses affines, comme celles données dans [6, 4]. 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.

Références

[1] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Proc. of FOCS'03*, pages 340–351, 2003.

[2] L. Bordeaux, M. Janota, J. P. Marques Silva, and P. Marquis. On unit-refutation complete formulae with existentially quantified variables. In *Proc. of KR'12*, 2012.

[3] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8) :677–692, 1986.

[4] Ch. De Cannière. Trivium : A stream cipher construction inspired by block cipher design principles. In *Proc. of ISC'06*, pages 171–186, 2006.

[5] J. Chen. XORSAT : An efficient algorithm for the dimacs 32-bit parity problem. *CoRR*, abs/cs/0703006, 2007.

[6] J.M. Crawford and M.J. Kearns. The minimal disagreement parity problem as a hard satisfiability problem. Technical report, 1995.

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

[8] A. Darwiche. SDD : A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI'11*, pages 819–826, 2011.

[9] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17 :229–264, 2002.

[10] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[11] Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *J. Artif. Intell. Res. (JAIR)*, 30 :565–620, 2007.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. of SAT'03*, pages 502–518. 2003.

[13] H. Fargier and P. Marquis. Extending the knowledge compilation map : Krom, Horn, affine and beyond. In *Proc. of AAAI'08*, pages 442–447, 2008.

[14] T. Frühwirth, L. Michel, and C. Schulte. Constraints in procedural and concurrent languages. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 24. Elsevier, 2006.

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

[16] M. Järvisalo, M. Heule, and A. Biere. Inprocessing rules. In *Proc. of IJCAR'12*, pages 355–370, 2012.

[17] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[18] C. Li. Equivalent literal propagation in the dll procedure. *Discrete Applied Mathematics*, 130(2) :251–276, 2003.

[19] M. L. Littman, S. M. Majercik, and T. Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning*, 27(3) :251–296, 2001.

[20] P. Marquis. Existential closures for knowledge compilation. In *Proc. of IJCAI'11*, pages 996–1001, 2011.

[21] R. Mateescu, R. Dechter, and R. Marinescu. AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. *Journal of Artificial Intelligence Research*, 33 :465–519, 2008.

[22] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.

[23] Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp : Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

[24] C. Piette, Y. Hamadi, and L. Saïs. Vivifying propositional clausal formulae. In *Proc. of ECAI'08*, pages 525–529, 2008.

[25] K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In *Proc. of AAAI'08*, pages 517–522, 2008.

[26] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2) :273–302, 1996.

[27] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.

[28] T. Sang, P. Beame, and H. A. Kautz. Performing Bayesian inference by weighted model counting. In *Proc. of AAAI'05*, pages 475–482, 2005.

[29] Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 475–482. AAAI Press / The MIT Press, 2005.

[30] Th. J. Schaefer. The complexity of satisfiability problems. In *Proc. of STOC'78*, pages 216–226, 1978.

[31] R. Schrag. Compilation for critically constrained knowledge bases. In *Proc. of AAAI'96*, pages 510–515, 1996.

[32] C.E. Shannon. The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28(1) :59–98, 1949.

[33] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT'09*, pages 244–257, 2009.

[34] S. Subbarayan, L. Bordeaux, and Y. Hamadi. Knowledge compilation properties of tree-of-BDDs. In *Proc. of AAAI'07*, pages 502–507, 2007.

[35] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8 :189–201, 1979.

[36] A. Voronov. *On Formal Methods for Large-Scale Product Configuration.* Ph.d. thesis, Chalmers University, 2013.

[37] M. Wachter and R. Haenni. Propositional DAGs : A new graph-based language for representing Boolean functions. In *Proc. of KR'06*, pages 277–285, 2006.