# Compacité pratique des diagrammes de décision valués

## Normalisation, heuristiques et expérimentations

**Hélène Fargier** [1], **Pierre Marquis** [2], **Nicolas Schmidt** [1 2]

*1. IRIT, 118 route de Narbonne, 31062 Toulouse Cedex*
*{fargier,schmidt}@irit.fr*

*2. CRIL, rue Jean Souvraz, 62307 Lens Cedex*
*marquis@cril.univ-artois.fr*

RÉSUMÉ. *Les diagrammes de décision valués (VDD) sont particulièrement intéressants pour la compilation de problèmes de satisfaction de contraintes valuées (VCSP). L'intérêt des différents langages de la famille VDD (en particulier, les langages ADD, SLDD, AADD) est qu'ils admettent des algorithmes en temps polynomial pour des traitements (comme l'optimisation) qui ne sont pas polynomiaux à partir des VCSP de départ. Comme l'efficacité pratique de tels traitements dépend de la taille du VDD compilé obtenu, il est important d'obtenir une forme la plus compacte possible. Nous décrivons dans cet article quelques résultats issus de nos travaux sur l'étude de la compacité pratique des VDD. Nous présentons un compilateur ascendant de VCSP en SLDD$_+$ (resp. SLDD$_\times$), un jeu d'heuristiques d'ordonnancement des variables, des procédures de traduction des langages SLDD$_+$ (resp. SLDD$_\times$) vers les langages ADD et AADD, et nous identifions quelques requêtes et transformations d'intérêt, réalisables en temps polynomial quand l'ensemble des affectations est représenté par un VDD. Les différents langages cibles et les heuristiques ont été testés sur deux familles de jeux d'essai, des VCSP additifs représentant des problèmes de configuration de voitures avec fonctions de coût, et des réseaux bayésiens. Il apparaît que, bien que le langage AADD soit strictement plus succinct en théorie que SLDD$_+$ (resp. SLDD$_\times$), le langage SLDD$_+$ (resp. SLDD$_\times$) convient bien en pratique quand il s'agit de compiler des problèmes de nature purement additive (resp. purement multiplicative).*

ABSTRACT. *Valued decision diagrams (VDDs) prove valuable data structures for compiling valued constraint satisfaction problems (VCSPs). Indeed, languages from the VDD family (especially, ADD, SLDD, AADD) benefit from polynomial-time algorithms for some tasks of interest (e.g., the optimization one) for which no polynomial-time algorithm exists when the input is the VCSP considered at start. Since the practical efficiency of such tasks depends in practice on the size of the compiled VDD, it is important to look for diagrams which are as compact as possible.*

*In this paper some results from our study of the practical compactness of VDDs are pointed out. We present a VCSP-to-SLDD$_+$ bottom-up compiler (resp. a VCSP-to-SLDD$_\times$ bottom-up compiler), several variable ordering heuristics, some translation procedures from SLDD$_+$ (resp. SLDD$_\times$) to ADD and to AADD, and we identify some useful queries and transformations than can be achieved in polynomial time when the set of assignments is represented as a VDD. The target languages and the heuristics under consideration have been tested on two families of benchmarks, additive VCSPs representing car configuration problems with cost functions and multiplicative VCSPs representing Bayesian nets. It turns out that even if the AADD language is strictly more succinct (from the theoretical side) than SLDD$_+$ (resp. SLDD$_\times$), the language SLDD$_+$ (resp. SLDD$_\times$) proves to be good enough in practice when purely additive (resp. purely multiplicative) problems are to be compiled.*

MOTS-CLÉS : *compilation, configuration, diagramme de décision valué.*

KEYWORDS: *compilation, configuration, valued decision diagram.*

## 1. Introduction

Les diagrammes de décision binaires – automates, diagrammes de décision binaires ordonnés (OBDD), diagrammes de décision multivalués (MDD) – sont particulièrement intéressants pour la compilation de problèmes de satisfaction de contraintes « dures ». En revanche, ils ne permettent pas de représenter directement des fonctions de coût, ou plus généralement des fonctions associant une valuation (coût, utilité, probabilité, etc.) aux affectations de variables. En d'autres termes, ils ne permettent pas la compilation de problèmes de satisfaction de contraintes *valuées* (VCSP). Pour un tel objectif, les diagrammes de décision valués (VDD) – diagrammes de décision algébriques (ADD) (Bahar *et al.*, 1993), diagrammes de décision à arcs valués (EVBDD) (Lai, Sastry, 1992 ; Lai *et al.*, 1996 ; Amilhastre *et al.*, 2002), *Semiring Labeled Decision Diagrams* (SLDD) (Wilson, 2005), diagrammes de décision algébriques affines (AADD) (Tafertshofer, Pedram, 1997 ; Sanner, McAllester, 2005) – sont des langages cibles pertinents. Les ADD par exemple ont été utilisés pour la compilation de problèmes de planification (Hoey *et al.*, 1999) ; des travaux en configuration de produit (Amilhastre *et al.*, 2002 ; Hadzic, Andersen, 2006) ont proposé d'utiliser les EVBDD (ou de manière équivalente, les SLDD additifs – SLDD$_+$) pour capturer des fonctions de coût ou de préférences fortement additives, c'est-à-dire des cas où la valuation associée à une affectation des variables s'exprime directement par un VCSP dont toutes les contraintes « souples » sont unaires : l'idée est alors de compiler les contraintes dures qui définissent le produit configurable par un MDD, puis d'ajouter les valuations définies par les contraintes unaires directement sur les arcs.

L'intérêt de ces langages de compilation est qu'ils permettent une gestion efficace de l'ensemble des solutions du VCSP, une fois compilé. La résolution interactive (c'est-à-dire par l'utilisateur) d'un problème de configuration avec fonction de coût par exemple se réduit à des opérations de conditionnement, de propagation et d'optimisation qui sont en temps linéaire dans la taille de la structure compilée. L'important

en pratique est donc d'obtenir une forme compilée la plus compacte possible. Plusieurs facteurs influencent la compacité des diagrammes de décision ordonnés :

– la compacité théorique (ou « *succinctness* ») qui est relative (elle indique s'il est possible de séparer exponentiellement ou non un langage d'un autre) et qui se focalise sur le pire cas ;

– la canonicité, c'est-à-dire la capacité de chaque langage à offrir pour chaque VCSP une forme canonique – ceci permet de reconnaître et de fusionner efficacement (par exemple, par un mécanisme de cache) les sous-diagrammes équivalents ;

– les heuristiques d'ordonnancement des variables choisies pour construire le diagramme de décision.

Les travaux de Sanner et McAllester (2005) ont montré que le langage AADD offre une forme canonique et est plus performant que le langage ADD du point de vue de la compacité théorique comme du point de vue pratique. Dans des travaux récents (Fargier *et al.*, 2013b), nous avons montré que la propriété de canonicité à l'œuvre dans les AADD pouvait être étendue, algébriquement, au langage SLDD, et qu'en théorie au moins, le langage AADD est plus succinct que SLDD$_+$ (resp. SLDD$_\times$), c'est-à-dire le langage des SLDD fondé sur le semi-anneau commutatif $\langle \mathbb{R}^+ \cup \{+\infty\}, 0, +\infty, +, \min \rangle$ (resp. le semi-anneau commutatif $\langle \mathbb{R}^+, 1, 0, \times, \max \rangle$).

Nous décrivons dans la suite quelques résultats issus de nos travaux portant sur l'étude de la compacité pratique des VDD. Nous présentons un compilateur ascendant de VCSP en SLDD$_+$ (resp. SLDD$_\times$), un jeu d'heuristiques d'ordonnancement des variables, des procédures de traduction du langage SLDD$_+$ (resp. SLDD$_\times$) vers les langages ADD et AADD, et nous identifions quelques requêtes et transformations d'intérêt réalisables en temps polynomial à partir de tels diagrammes valués. Les différents langages cibles et les heuristiques ont été testés sur deux familles de jeux d'essai, des VCSP additifs représentant des problèmes de configuration de voitures avec fonctions de coût, et des réseaux bayésiens. Il apparaît que, quoique le langage AADD soit strictement plus succinct en théorie que SLDD$_+$ et que SLDD$_\times$, le langage SLDD$_+$ (resp. SLDD$_\times$) convient bien en pratique quand il s'agit de compiler des problèmes de nature purement additive (resp. purement multiplicative).

## 2. Diagrammes de décision valués

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble de variables où chaque $x_i \in X$ prend ses valeurs dans un domaine fini et discret $D_x$ ; on note $D_X$ l'ensemble des affectations $\vec{x}$ des variables de $X$. Un diagramme de décision $\alpha$ est une structure de données permettant de représenter une fonction $f_\alpha$ qui associe à chaque affectation $\vec{x} = \{(x_i, d_i) \mid d_i \in D_{x_i}, i = 1, \ldots, n\}$ un élément d'un ensemble $E$ de *valuations*. $E$ est le support d'une structure de valuation $\mathcal{E}$ qui peut être plus ou moins riche d'un point de vue algébrique. Dans le formalisme ADD, aucune hypothèse n'est faite sur $E$ (bien que l'on considère généralement que $E = \mathbb{R}$). Pour le langage AADD, $E = \mathbb{R}^+$.

DÉFINITION 1 (Diagramme de décision valué). — *Un* diagramme de décision valué *(VDD) est un graphe orienté et acyclique avec une seule racine, où chaque nœud $N$ est étiqueté par une variable $Var(N) = x$ où $x \in X$; si $D_x = \{d_1, \ldots, d_k\}$, alors $N$ a $k$ arcs sortants $a_1, \ldots, a_k$, tels que chaque $a_i$ est étiqueté par la valeur $val(a_i) = d_i$. Les variables étiquetant les nœuds de tout chemin de la racine à une feuille sont toutes distinctes. Les nœuds $N$ (resp. les arcs $a$) peuvent également être étiquetés par une valuation $\phi(N)$ (resp. $\phi(a)$) de $E$. On note $In(N)$ (resp. $Out(N)$) l'ensemble des arcs entrant dans (resp. issus de) $N$ et $In(a)$ (resp. $Out(a)$) le nœud origine (resp. l'extrêmité) de $a$.*

Ces diagrammes sont généralement *ordonnés* : un ordre total $<$ sur $X$ est choisi et l'on impose que la suite des variables associées aux nœuds rencontrés sur chaque chemin de la racine vers une feuille soit compatible avec cet ordre.

Un diagramme de décision valué est dit *sous forme réduite* s'il ne contient pas de nœuds isomorphes (deux nœuds étiquetés par la même variable et dont les arcs sortants sont identiques, c'est-à-dire pointent sur les mêmes nœuds en portant la même valeur du domaine de leur variable et la même valuation). Tout VDD ordonné possède une unique forme réduite, qu'il est possible d'obtenir en temps linéaire en sa taille, soit par une procédure de réduction remontant de la/des feuille/s vers la racine, soit simplement par un mécanisme de cache (une « table d'unicité »). Dans la suite, nous supposons que les diagrammes de décision considérés sont sous forme réduite.

Le langage ADD est la généralisation aux valuations non booléennes du langage OBDD, les deux nœuds terminaux 0 et 1 des OBDD étant remplacés par autant de nœuds que de valeurs de $E$ associées à une affectation au moins.

DÉFINITION 2 (ADD). — *Un ADD est un VDD ordonné dont seuls les nœuds terminaux sont valués (les arcs ne le sont pas). Un ADD $\alpha$ associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in E$ définie par :*

– *si $\alpha$ est un nœud terminal $N$ , alors $f_\alpha(\vec{x}) = \phi(N)$*

– *sinon, la racine $N$ de $\alpha$ est étiquetée par $x \in X$. Soient $d$ la valeur de $x$ dans $\vec{x}$, $a = (N, M)$ l'arc issu de $N$ tel que $val(a) = d$, et $\beta$ le ADD de racine $M$ dans $\alpha$ ; on a pour tout $\vec{x} \in D_X$ : $f_\alpha(\vec{x}) = f_\beta(\vec{x})$.*

Les nœuds terminaux reprenant l'ensemble des valeurs possibles de la fonction représentée, le nombre de nœuds d'un ADD croît avec le cardinal de l'ensemble de ces valeurs (l'image de la fonction représentée). Ainsi, la fonction $f(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} 2^{i-1} x_i$ sur $\{0, 1\}^n$, qui est représentable en espace polynomial par un VCSP fortement additif, prend $2^n$ valeurs différentes, d'où une taille exponentielle des ADD qui la représentent.

Dans les SLDD (*Semiring Labeled Decision Diagrams*) tels que définis dans (Wilson, 2005), la structure de valuation est un semi-anneau $\mathcal{E} = \langle E, \otimes, \oplus, 1_s, 0_s \rangle$, $1_s$ dénotant l'élément neutre de l'opérateur $\otimes$ et $0_s$ dénotant l'élément neutre de l'opérateur $\oplus$, absorbant pour $\otimes$. L'opérateur $\oplus$ n'a aucune influence pour la définition d'un SLDD en tant que représentation d'une fonction de $D_X$ dans $E$ ($\oplus$ est utilisé lorsque l'on

veut calculer une valuation optimale, ou lorsque l'on veut éliminer une ou plusieurs variables). Pour cette raison, nous utilisons dans la suite une définition un peu plus générale que celle de (Wilson, 2005), exigeant simplement une structure de monoïde pour $\mathcal{E} = \langle E, \otimes, 1_s \rangle$ : $\otimes$ est une loi interne à $E$, associative, et qui possède un élément neutre $1_s$ (Fargier *et al.*, 2013b).

DÉFINITION 3 (SLDD). — *Un SLDD $\alpha$ sur $X$ est un VDD avec une unique racine et un unique nœud terminal, dont les arcs sont étiquetés par des éléments de $E$ où $\mathcal{E} = \langle E, \otimes, 1_s \rangle$ est un monoïde. Un SLDD associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x})$ appartenant à $E$ définie par :*

– *si $\alpha$ est le nœud terminal alors $f_\alpha(\vec{x}) = 1_s$*

– *sinon, la racine $N$ de $\alpha$ est étiquetée par $x \in X$. Soient $d \in D_x$ la valeur de $x$ dans $\vec{x}$, $a = (N, M)$ l'arc issu de $N$ tel que $val(a) = d$, et $\beta$ le SLDD de racine $M$ dans $\alpha$ ; on a pour tout $\vec{x} \in D_X$ : $f_\alpha(\vec{x}) = \phi(a) \otimes f_\beta(\vec{x})$.*

*À des fins de normalisation, on peut associer à $\alpha$ une valeur $\phi_0 \in E$ (son offset). La fonction « augmentée » $f_{\alpha,\phi_0}$ que représente $\alpha$ ainsi décoré par $\phi_0$, est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\phi_0}(\vec{x}) = \phi_0 \otimes f_\alpha(\vec{x})$.*

Deux monoïdes sont particulièrement intéressants : $\mathcal{E} = \langle \mathbb{R}^+ \cup \{+\infty\}, +, 0 \rangle$ pour tous les problèmes dont les valuations sont de nature additive (coûts, utilités, ...) et $\mathcal{E} = \langle \mathbb{R}^+, \times, 1 \rangle$ pour tous les problèmes dont les valuations sont de nature multiplicative (probabilités). Les langages associés sont notés respectivement SLDD$_+$ et SLDD$_\times$. Chacun admet un élément absorbant ($+\infty$ pour les SLDD$_+$, 0 pour les SLDD$_\times$), ce qui permet de compiler des VCSP possédant des contraintes « dures » : dans un SLDD$_+$ par exemple, toute affectation $\vec{x}$ telle que $f(\alpha)(\vec{x}) = +\infty$ est considérée comme non admissible car violant une contrainte « dure ».

Dans la suite, nous supposerons les SLDD ordonnés selon un ordre $<$ sur $X$ fixé. Cela est nécessaire d'une part pour pouvoir les comparer aux AADD et aux ADD qui sont des structures ordonnées, et d'autre part parce que cette propriété garantit une forme réduite canonique (et des opérations de combinaison en temps polynomial).

Enfin, les diagrammes de décision algébriques affines introduits dans (Tafertshofer, Pedram, 1997 ; Sanner, McAllester, 2005) permettent d'utiliser conjointement les opérateurs $\times$ et $+$ sur $\mathbb{R}^+$. Dans un SLDD, un arc $a$ porte une simple valuation $\phi(a)$ alors que dans un AADD, les arcs sont étiquetés par des *couples* de valeurs.

DÉFINITION 4 (AADD). — *Un AADD $\alpha$ sur $X$ est un VDD ordonné avec une unique racine et un unique nœud terminal, dont les arcs sont étiquetés par des couples d'éléments de $\mathbb{R}^+$. $\alpha$ associe à toute $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in \mathbb{R}^+$ définie par :*

– *si $\alpha$ est le nœud terminal $N$, $f_\alpha(\vec{x}) = 0$*

– *sinon, la racine $N$ de $\alpha$ est étiquetée par $x \in X$. Soient $d \in D_x$ la valeur de $x$ dans $\vec{x}$, $a = (N, M)$ l'arc issu de $N$ tel que $val(a) = d$, $\phi(a) = \langle q, f \rangle$ le couple de valeurs associée à $a$ et $\beta$ le AADD de racine $M$ dans $\alpha$ ; on a pour tout $\vec{x} \in D_X$ : $f_\alpha(\vec{x}) = q + (f \times f_\beta(\vec{x}))$.*

*À des fins de normalisation, on attache à $\alpha$ un couple $\langle q_0, f_0 \rangle$ de $\mathbb{R}^+ \times \mathbb{R}^+$ (son offset). La fonction « augmentée » $f_{\alpha,\langle q_0,f_0\rangle}$ que représente $\alpha$ ainsi décoré par $\langle q_0, f_0 \rangle$, est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\langle q_0,f_0\rangle}(\vec{x}) = q_0 + (f_0 \times f_\alpha(\vec{x}))$.*

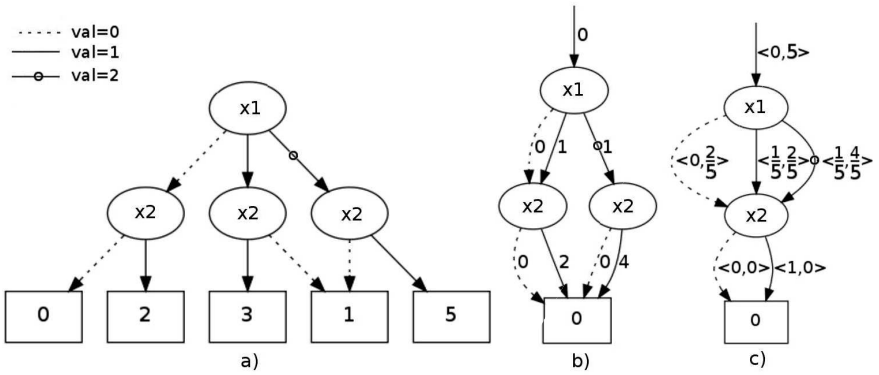


*Figure 1. Exemple d'un ADD (a) d'un SLDD$_+$ (b) et d'un AADD (c) représentant la même fonction. $D_{x_1} = \{0, 1, 2\}, D_{x_2} = \{0, 1\}$*

Grosso modo, un SLDD$_+$ peut être considéré comme un AADD particulier dont les facteurs multiplicatifs $f$ portés par les derniers arcs (aboutissant au puits) sont égaux à 0 alors que tous les autres sont égaux à 1. De la même façon, un SLDD$_\times$ peut être grosso modo considéré comme un AADD particulier dont les facteurs additifs $q$ sont nuls [1]. La figure 1 décrit (a) un ADD (b) un SLDD$_+$ et (c) un AADD, tous les trois représentant la même fonction.

## 3. Normalisation

Le langage AADD est muni d'une procédure de normalisation qui permet de garantir, pour chaque fonction à représenter et étant donné un ordre des variables, une unique représentation réduite. Cette propriété de canonicité joue un rôle important dans l'obtention de forme compacte, puisqu'elle permet de détecter, et donc de fusionner, les sous-diagrammes qui représentent la même fonction.

### *3.1. Normalisation des AADD (Sanner, McAllester, 2005)*

DÉFINITION 5 (Normalisation d'un AADD). — *Soient un nœud $N$ étiqueté par une variable $x$ dont le domaine contient $n$ valeurs, $Out(N) = \{a_1, a_2, \ldots, a_n\}$ l'ensemble des arcs sortants de $N$, et pour chaque $i \in 1, \ldots, n$, $\phi(a_i) = \langle q_i, f_i \rangle$ la valuation attachée à cet arc. $N$ est* normalisé *ssi :*

1. Pour plus de détails sur la correspondance entre AADD et SLDD, se référer à la section 4.3

**Algorithme 1 :** *NormaliseAADD*$(\alpha, \langle q_0, f_0 \rangle)$

input : Un AADD $\alpha$ et son offset $\langle q_0, f_0 \rangle$
output : Un AADD normalisé équivalent à $\alpha$

1 for *each node* $N$ *of* $\alpha$ *in inverse topological ordering* do
2 $\quad q_{min} \leftarrow \min_{a \in Out(N)} q_a$;
3 $\quad range \leftarrow \max_{a \in Out(N)}(q_a + f_a) \; - \; q_{min}$;
4 $\quad$ for *each* $a \in Out(N)$ do
5 $\quad\quad$ if $range > 0$ then
6 $\quad\quad\quad q_a \leftarrow (q_a - q_{min})/range$;
7 $\quad\quad\quad f_a \leftarrow f_a/range$;
8 $\quad\quad$ else // Ici, $f_a = 0$ et $q_a = q_{min}$
9 $\quad\quad q_a \leftarrow q_a - q_{min}$;
10 $\quad\quad$ ;
11 $\quad$ for *each* $a \in In(N)$ do
12 $\quad\quad q_a \leftarrow q_a + q_{\min} \times range$;
13 $\quad\quad f_a \leftarrow f_a \times q_{range}$;
14 $q_0 \leftarrow q_0 + q_{min} \times range$;
15 $f_0 \leftarrow f_0 \times range$;
16 return $\alpha$;

– $min(q_1, q_2, ..., q_n) = 0$

– $max(q_1 + f_1, q_2 + f_2, ..., q_n + f_n) = 1$

– *pour chaque arc* $a_i = (N, M_i)$*, l'AADD* $\beta_i$ *enraciné en* $M_i$ *est tel que si* $\forall \vec{x} \in D_X$, $f_{\beta_i}(\vec{x}) = 0$*, alors* $f_i = 0$.

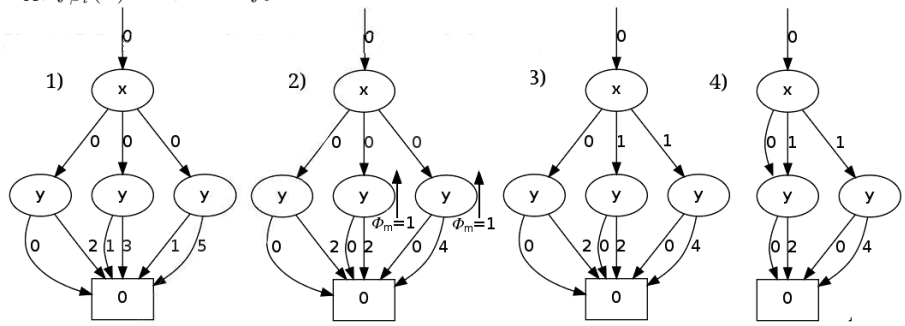


*Figure 2. Exemple de normalisation d'un SLDD$_+$. 1) SLDD$_+$ non normalisé. 2) SLDD$_+$ obtenu après exécution des instructions aux lignes 2 à 5. 3) SLDD$_+$ obtenu après exécution des instructions aux lignes 6 et 7. 4) SLDD$_+$ normalisé sous forme réduite après fusion des deux nœuds isomorphes*

Tout AADD peut être normalisé en temps linéaire (voir algorithme 1). Cette procédure normalise les nœuds $N$ du puits vers la racine de la structure. Pour chaque $N$,

**Algorithme 2 :** *NormaliseSLDD*$(\alpha, \phi_0)$

input : Un SLDD $\alpha$, et son offset $\phi_0$
output : Un SLDD normalisé équivalent à $\alpha$
// For SLDD$_+$ let
// $\otimes = +, \otimes^{-1} = -, \oplus = min, 1_s = 0$
// For SLDD$_\times$ let
// $\otimes = \times, \otimes^{-1} = \oplus = max, 1_s = 1$

```
for each node N of α in inverse topological ordering do
    φ_m ← ⊕_{a∈Out(N)} φ(a);
    // Rem : quand ⊕ = max, φ_m = 0 ssi
    // ∀a, φ(a) = 0
    for each a ∈ Out(N) do
        if φ(a) = φ_m then φ(a) ← 1_s;
        else φ(a) ← φ(a) ⊗^{-1} φ_m;
    for each a ∈ In(N) do
        φ(a) ← φ(a) ⊗ φ_m;
φ_0 ← φ_0 ⊗ φ_m;
return α;
```

elle calcule un « offset interne » $\langle q_{min}, range \rangle$, qui permet de normaliser $N$ et puis est reporté sur les parents de $N$, ou sur l'offset de l'AADD si $N$ est sa racine. Appliquée depuis le puits vers la racine, cette procédure remonte sur l'offset du diagramme les valeurs minimum ($q_0$) et maximum ($q_0 + f_0$) de la fonction représentée par l'AADD.

### *3.2. Normalisation des SLDD*

Les SLDD$_+$ et SLDD$_\times$ pouvant être vus comme des AADD, nous nous appuyons sur l'existence d'une forme normalisée pour les AADD pour proposer une définition similaire (quoique plus simple) d'une forme normalisée pour les SLDD$_+$ et SLDD$_\times$.

DÉFINITION 6 (Normalisation d'un SLDD). —

– *Un SLDD$_+$ est normalisé ssi pour tout nœud $N$ d'arcs sortants $Out(N) = \{a_1, a_2, \ldots, a_n\}$, $min(\phi(a_1), \phi(a_2), ..., \phi(a_n)) = 0$.*

– *Un SLDD$_\times$ est normalisé ssi pour tout nœud $N$ d'arcs sortants $Out(N) = \{a_1, a_2, \ldots, a_n\}$, $max(\phi(a_1), \phi(a_2), ..., \phi(a_n)) = 1$.*

On peut utiliser une spécialisation de la procédure de (Sanner, McAllester, 2005) pour normaliser un SLDD (voir algorithme 2). Dans un SLDD$_+$ (resp. SLDD$_\times$) normalisé, le minimum (resp. le maximum) de la fonction représentée est ainsi remonté à la racine en offset.

Nous utilisons une « table d'unicité » (i.e., une table de hachage) pour le stockage des nœuds. Celle-ci permet de détecter en temps constant (quand le nombre d'entrées est borné) si le nœud que l'on vient de normaliser est isomorphe à un nœud déjà existant. La clé de hachage d'un nœud $N$ est donc calculée en fonction de la variable $Var(N)$, des valuations $\phi(a)$ portées par les arcs $a$ sortant de $N$ et de l'ensemble des nœuds $M$ pointés par ces arcs. Ainsi, un SLDD normalisé est automatiquement réduit sous sa forme canonique.

## 4. Construction de diagrammes de décision valués

À des fins expérimentales, nous avons implémenté un compilateur de VDD. Ce compilateur permet d'une part de compiler des VCSP à valuations additives (Bistarelli *et al.*, 1999) dont les contraintes sont exprimées par des tables valuées, i.e., chaque $n$-uplet $t$ a un attribut valuation $t[\phi]$; ces tables sont représentés selon le format XCSP 2.1 décrit dans (Roussel, Lecoutre, 2009) et compilées sous la forme de SLDD$_+$. D'autre part, notre compilateur permet de compiler des réseaux bayésiens (selon le format XML de (Cozman, 2002)) sous la forme de SLDD$_\times$. Pour permettre l'obtention de différents types de VDD, nous avons également implémenté des algorithmes pour la traduction de SLDD$_+$ et SLDD$_\times$ en ADD et réciproquement, ainsi que des algorithmes pour la traduction de SLDD$_+$ et SLDD$_\times$ en AADD.

### *4.1. Compilation*

Notre approche de la compilation d'un VCSP en SLDD suit un procédé ascendant classique pour la construction de diagrammes de décisions ordonnés, valués ou non (Sanner, McAllester, 2005 ; Bryant, 1986 ; Amilhastre, 1999).

Nous déterminons tout d'abord un ordre total $<$ des variables selon lequel le SLDD à construire sera ordonné (la section suivante décrit et compare les heuristiques d'ordonnancement de variables). On crée ensuite un SLDD « blanc », c'est-à-dire un SLDD de $n + 1$ nœuds, avec $n$ le nombre de variables, où l'on associe à chaque variable un nœud dont chacun des arcs sortants $a$ (il y en a un par valeur dans le domaine de la variable) pointe vers le nœud portant la variable suivante selon l'ordre $<$ avec une valuation $\phi(a) = 0$ (ou $\phi(a) = 1$ pour un SLDD$_\times$). Les contraintes sont ensuite ajoutées une à une au SLDD en construction.

#### *Ajout de contraintes*

Nous décrivons ici la procédure d'ajout d'une contrainte valuée à un SLDD$_+$. L'ajout d'une contrainte à un SLDD$_\times$ suit le même schéma, en remplaçant toute addition par une multiplication, toute soustraction par une division, et toute occurrence de l'élément neutre $0$ par une occurrence de l'élément neutre $1$.

Chaque contrainte $c$ est représentée par une table $T$ qui porte sur un ensemble $Var(T) \subseteq X$ de variables. Si $c$ est une contrainte « souple », elle associe une valuation $t[\phi] \in E$ à chaque $n$-uplet $t$ de $T$, et une valuation par défaut $\phi_0$ aux affectations

**Algorithme 3 :** *AjouteContrainte*$(T, \alpha, \phi_0)$

input : Un SLDD ordonné $\alpha$, une table de $n$-uplets $T$ valués, une valuation par défaut $\phi_0$

output : Le SLDD $\alpha$ modifié de manière à représenter la fonction $f_\alpha \otimes f_T$

```
1 x ← first(Var(T));
2 for all nœuds M tels que Var(M) = x do
3     for all arcs a_i ∈ Out(M) do
4         T'_i ← {n-uplets de T tels que t[x] = val(a_i)};
5         AjouteContrainteArc(T'_i, a_i, φ_0);
```

non explicitement présentes dans la table. Si $c$ est une contrainte « dure », on associe l'élément neutre de $\mathcal{E}$ (0 pour un SLDD$_+$, 1 pour un SLDD$_\times$) aux $n$-uplets qui la satisfont, et $\phi_0$ est égal à l'élément absorbant de $\mathcal{E}$ ($+\infty$ pour un SLDD$_+$, 0 pour un SLDD$_\times$). On suppose que les variables de la contrainte sont ordonnées selon le même ordre que celui utilisé dans le SLDD. Il est toujours possible de voir chaque contrainte $c$ représentée par $T$ comme un ADD, ou comme un SLDD non réduit dont seuls les derniers arcs portent une valuation différente de l'élément neutre de $\mathcal{E}$. On note $f_T$ la fonction correspondant à $T$ associant à chaque $\vec{x} \in D_X$ la valuation $t[\phi] \in E$ telle que $\forall x_i \in Var(T), t[x_i] = x_i$ quand un tel $n$-uplet $t \in T$ existe, et $\phi_0$ sinon.

L'algorithme 3-4 effectue l'opération de fusion selon l'opérateur $\otimes$ considéré (l'addition pour un SLDD$_+$, la multiplication pour un SLDD$_\times$) entre une table de $n$-uplets $T$ et un SLDD $\alpha$. L'algorithme 3 permet de ne commencer l'opération qu'à partir de la première variable de $Var(T)$. Il se contente simplement de sélectionner les nœuds correspondants à cette variable, et de lancer la fonction AjouteContrainteArc$(T, a, \phi_0)$ sur chacun des arcs de $\alpha$ issus d'un nœud étiqueté par cette variable. La fonction récursive AjouteContrainteArc est inspirée de l'algorithme Apply proposé dans (Bryant, 1986) (algorithme qui combine deux OBDD selon un opérateur logique) et repris par (Sanner, McAllester, 2005) : il s'agit de combiner (selon $\otimes$) un SLDD$_\otimes$ ($\alpha$) et une fonction associant une valeur $t[\phi]$ à chaque $n$-uplet $t$ de $T$ (ou la valeur $\phi_0$ sinon) - un pseudo SLDD, comme suggéré plus haut ; comme chez (Bryant, 1986), les arcs des deux structures sont suivis en parallèle. Dans le pseudo SLDD, seuls les arcs issus des derniers nœuds, les arcs finaux des $n$-uplets $t$, portent une valuation différente de l'élément neutre. Par conséquent, la combinaison de valuations ne s'opère que lorsque tout le chemin correspondant à un $n$-uplet $t$ est parcouru.

De plus, un nœud $M$ pouvant être atteint plusieurs fois lors du traitement de la table $T$ (en particulier parce que $T$ n'affecte pas toutes les variables du SLDD), la procédure Visite (algorithme 5) permet de détecter si un nœud a déjà été rencontré (i.e., si la procédure a déjà été appelée avec les mêmes valeurs des paramètres $T$ et $M$), et quel est le nœud qui a résulté de cette visite : si c'est le cas, on retourne ce nœud résultant, sinon on le crée et on le mémorise dans la table d'unicité $VisitePrecedente$.

**Algorithme 4 :** *AjouteContrainteArc*$(T, a, \phi_0)$

```
input  : Un arc a d'un SLDD ordonné α, une table de n-uplets T valués, une
         valuation par défaut φ_0
output : Le SLDD α modifié de manière à représenter la fonction f_α ⊗ f_T
// ⊗ = × for SLDD_×; ⊗ = + for SLDD_+

// si aucun n-uplet dans T
1  if T = ∅ then
2  |  φ(a) ← φ(a) ⊗ φ_0;
   // si l'ensemble des variables de Var(T) a été
      parcouru
3  else if Var(In(a)) = last(Var(T)) then
   |  // ici, il ne reste plus qu'un seul tuple t dans T
4  |  φ(a) ← φ(a) ⊗ t[φ];
5  else
6  |  M ← Visite(T, a);
   |  // rediriger l'arc a sur M
7  |  Out(a) ← M;
8  |  if Var(M) ∈ Var(T) then
9  |  |  for all arcs a' ∈ Out(M) do
10 |  |  |  T' ← {n-uplets t de T tel que t[Var(M)] = val(a')};
11 |  |  |  AjouteContrainteArc(T', a', φ_0);
12 |  else
13 |  |  for all arcs a' ∈ Out(M) do
14 |  |  |  AjouteContrainteArc(T, a', φ_0);
```

Pour ne pas alourdir l'algorithme, nous n'avons pas detaillé les actions de normalisation. En pratique, la (re)normalisation du SLDD construit est réalisée « à la volée », c'est-à-dire que dès qu'un nœud $M$ a été traité (juste avant de quitter la procédure récursive), il est directement normalisé et l'offset calculé est remonté sur les arcs entrants dans ce nœud (par construction, ces arcs sont issus de nœuds qui n'ont pas encore été introduits dans la table d'unicité ; il le seront au retour de la procédure récursive). Puis $M$ est introduit dans la table d'unicité, si aucun nœud de même clé (donc isomorphe) n'y est présent. Si $M$ est déjà présent, il est remplacé par le nœud contenu dans la table. À la fin du traitement, on normalise le SLDD selon la procédure NormaliseSLDD (algorithme 2), en partant non du puits, mais des nœuds étiquetés par la première des variables de $T$ selon l'ordre considéré (les nœuds d'où a été lancée la procédure, dans l'algorithme AjouteContrainte), puis leurs prédécesseurs sont (re)normalisés à leur tour si besoin est.

**Algorithme 5 :** *Visite*$(T, a)$

input : Un arc $a$ et une table $T$ de $n$-uplets valués
output : Un nœud résultant de la visite de $Out(a)$ après prise en compte de $T$

```
// Utilise une table VisitePrecedente (variable globale
   et rémanente) qui à toute paire (M,T) associe le
   nœud résultant de la visite de M étant donné T,
   ou null si aucune visite n'a encore été effectuée
```

$M \leftarrow Out(a)$;
// Le nœud $M$ n'a jamais été atteint lors du traitement de $T$
if $VisitePrecedente(M, T) = null$ then
  // si $a$ est l'unique arc entrant de $M$
  if $In(M) = \{a\}$ then
    $M' \leftarrow M$;
  else
    $M' \leftarrow \mathsf{Copie}(M)$;
  // Mémoriser $M'$ comme étant le résultat de la visite de $M$ étant donné $T$
  $VisitePrecedente(M, T) \leftarrow M'$;
return $M'$;

### *4.2. Heuristiques*

La taille des diagrammes de décision binaires est très sensible à l'ordre dans lequel ses variables ont été ordonnées (Bryant, 1986 ; Amilhastre, 1999 ; Drechsler, 2002) – et c'est également le cas pour leurs versions valuées. Nous avons appliqué aux VDD plusieurs heuristiques proposées pour les MDD (Amilhastre, 1999 ; Drechsler, 2002).

MCF (Amilhastre, 1999)

Afin de réduire la taille du diagramme de décision, il peut sembler intéressant de rencontrer au plus tôt les variables intervenant dans le plus de contraintes. On espère ainsi, lorsque le problème contient des contraintes dures, traiter le plus tôt possible des arcs portant la valeur absorbante : ces arcs joignant directement le puits, cela limite la taille du VDD. L'heuristique *Most Constrained First* (MCF) trie les variables en fonction du nombre de contraintes « dures » dans lesquelles elles sont utilisées. Les variables sont donc d'autant plus prioritaires que la valeur $MCF$ est haute :

$$MCF(x) = |\{c \mid x \in Var(c)\}|.$$

Band-Width (Amilhastre, 1999)

(Amilhastre, 1999) a montré qu'une heuristique fondée sur la *Band-Width* du graphe de contraintes permet de borner la taille du meilleur MDD décrivant un CSP classique.

Soit $O : \{1, \ldots, n\} \mapsto X$ un ordre sur les $n$ variables de $X$ ($O$ associe à chaque rang une variable). La *Band-Widt*h d'un ordre $O$ est :

$$BW(O) = max_{i,j=1}^{n}\{j - i \mid i < j \text{ et } O[i] \text{ voisine}^{2} \text{ de } O[j]\}.$$

Le calcul d'un ordre de *Band-Width* minimum pour un graphe quelconque étant un problème NP-difficile (Amilhastre, 1999), nous utilisons un algorithme glouton pour l'approcher. On choisit les variables de l'ordre itérativement : la première sélectionnée (celle qui sera en tête du VDD) est une des variables les plus contraintes. Étant donnée une suite $O$ de $k$ variables sélectionnées, la variable suivante (au rang $k + 1$) est la variable $x$ non encore sélectionnée qui maximise la quantité :

$$H_O(x) = max_{i=1}^{k}\{k + 1 - i \mid O[i] \text{ voisine de } x\}.$$

MCS-Inv (Tarjan, Yannakakis, 1984)

MCS (pour *Maximum Cardinal Search*) est une méthode introduite dans le cadre de reconnaissance de graphes triangulés. La méthode MCS-Inv reprend le même principe en inversant l'ordre final. Elle permet aux variables fortement contraintes d'être proches des variables avec lesquelles elles sont liées. Notre implémentation utilise un algorithme glouton. On choisit les variables de l'ordre itérativement : la première sélectionnée est la variable la plus contrainte. Étant donnée une suite $O$ de $k$ variables sélectionnées, la variable suivante (au rang $k + 1$) est la variable $x$ non encore sélectionnée qui maximise la quantité :

$$MCS_O(x) = \Sigma_{i=1}^{k} \ \{k + 1 - i \mid O[i] \text{ voisine de } x\}.$$

Force (Aloul *et al.*, 2003)

Le but de *Force* est de minimiser le *span* induit sur le graphe de contraintes, c'est-à-dire la somme (et non pas, comme pour l'heuristique *Band-Width*, le maximum) des distances séparant les variables voisines.

$$span(O) = \Sigma_{i,j=1}^{n}\{j - i \mid i < j \text{ et } O[i] \text{ voisine de } O[j]\}.$$

La méthode consiste à calculer, à partir d'un ordre quelconque sur les variables le « centre de gravité ($COG$) » de chacune des contraintes $c \in C$ :

$$COG(c) = \frac{\Sigma_{x \in Var(c)} POS(x)}{|Var(c)|}.$$

Ici, $POS(x)$ est la position de $x$ dans l'ordre courant. On remet alors à jour les positions des variables en fonction des centres de gravité des contraintes auxquels elles appartiennent :

2. Deux variables $x$ et $y$ sont dites voisines si elles appartiennent à une même contrainte, c'est-à-dire s'il existe $c \in C$ tel que $x \in Var(c)$ et $y \in Var(c)$.

$$POS(x) = \frac{\Sigma_{c|x \in Var(c)} COG(c)}{|\{c|x \in Var(c)\}|}.$$

Cette procédure part d'un ordre quelconque $O$ sur les variables : à l'initialisation, $POS(x)$ est le rang de $x$ dans $O$. Elle est répétée autant de fois que nécessaire, jusqu'à arriver à un point fixe, où, d'une itération à l'autre, les estimations de centres de gravité des variables n'évoluent plus. On réordonne ensuite les variables par $POS$ croissant.

### *4.3. Traductions ADD⇌SLDD⇌AADD*

Nous développons ici les principes qui fondent les traductions d'un type de VDD vers chacun des autres (en supposant que tous partagent le même domaine de valuation, typiquement $E = \mathbb{R}^+ \cup \{+\infty\}$, et le même ordre sur les variables). Ces traductions peuvent souvent être vues comme l'application de procédures de normalisation.

*Traductions SLDD↦AADD, ADD↦SLDD, ADD↦AADD*

La traduction la plus évidente est la traduction d'un SLDD$_+$ (ou d'un SLDD$_\times$) en AADD : tout SLDD$_+$ peut être transformé en AADD en remplaçant, pour chaque arc $a$ à destination du puits sa valuation $\phi(a)$ par le couple $\langle \phi(a), 0 \rangle$, et pour tout autre arc $a'$, sa valuation $\phi(a')$ par le couple $\langle \phi(a'), 1 \rangle$. On normalise ensuite l'AADD obtenu, ce qui permet éventuellement de le réduire. De la même façon, tout SLDD$_\times$ peut être transformé en AADD en remplaçant, pour chaque arc $a$, son étiquette $\phi(a)$ par le couple $\langle 0, \phi(a) \rangle$. On normalise et réduit ensuite l'AADD obtenu, ce qui permet éventuellement d'obtenir une structure plus compacte que le diagramme original.

De la même façon, on peut toujours transformer un ADD en SLDD$_+$ ou en SLDD$_\times$ en reportant les valuations portées par les nœuds terminaux sur leurs arcs entrants (les autres arcs portant la valuation $0$ lorsque l'on veut construire un SLDD$_+$ la valuation $1$ lorsque l'on veut construire un SLDD$_\times$) les nœuds terminaux sont remplacés par le puits du nouveau SLDD. On normalise et réduit ensuite le diagramme obtenu selon les principes de normalisation des SLDD$_+$ (resp. des SLDD$_\times$).

La transformation d'un ADD en AADD, est similaire, les valuations $\phi_i$ des nœuds terminaux étant reportées sur leurs arcs entrants sous la forme d'un couple $\langle 0, \phi_i \rangle$.

*Traductions SLDD↦ADD, AADD↦ADD, AADD↦SLDD*

Transformer un SLDD en ADD revient à repousser les valuations $\phi$ vers les arcs du dernier niveau. En quelque sorte, il s'agit d'une normalisation assurant que pour tout $a$, $\phi(a)$ est égal à l'élément neutre de $\mathcal{E}$ ($0$ pour les SLDD$_+$ et $1$ pour les SLDD$_\times$) il faut alors, pour porter les valuations, copier le puits en autant de nœuds finaux que de valuations différentes sur ses arcs entrants. La procédure de « normalisation » du SLDD en ADD procède de la racine vers le puits plus précisément, pour tout nœud $N$ dont les parents ont été traités :

– remplacer $N$ par autant de copies $N_1, \ldots N_k$ de $N$ que de valuations différentes $\phi_1, \ldots, \phi_k$ sur les arcs entrant dans $N$ ;

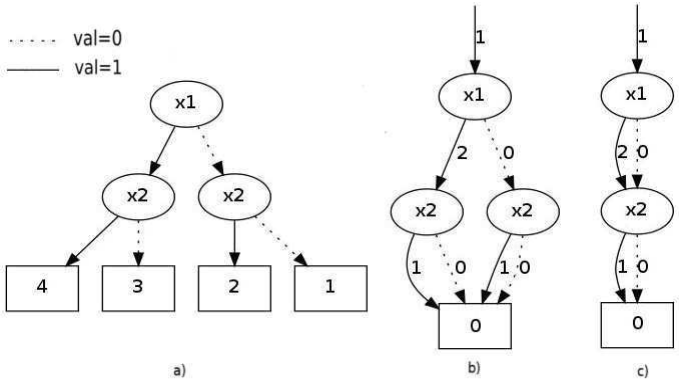


*Figure 3. Exemple de traduction d'un ADD (a) en SLDD$_+$ : en (b) les poids sont remontés sur les arcs et les nœuds sont normalisés ; en (c) les nœuds isomorphes sont fusionnés*

– si $N$ est le puits du diagramme, chaque copie $N_i$ porte la valuation $\phi_i$ ;

– sinon ($N$ est un nœud interne), pour chaque copie $N_i$, et chaque arc $a$ de $N$ vers un nœud $M$, créer un arc $a'$ de $N_i$ vers $M$, étiqueté par $val(a)$ et prenant pour valuation $\phi(a') = \phi(a) \otimes \phi_i$

– normaliser et réduire le diagramme en fusionnant les nœuds isomorphes.

Une procédure similaire est appliquée pour transformer un AADD en SLDD$_+$ : on crée autant de copies $N_1, \ldots, N_k$ de $N$ que de facteurs multiplicatifs différents $f_1$, …, $f_k$ sur les arcs $a_j$ entrant dans $N$. Chaque copie $N_i$ est liée aux prédécesseurs de $N$ par des copies $a'$ des arcs $a$ entrant dans $N$ et portant la valuation $f_i$ : chaque arc $a'$ porte la même valeur du domaine de sa variable que $a$ et sa valuation est $\langle q_a, 1\rangle$ le facteur multiplicatif de $a$ est reporté sur les arcs suivants : pour chaque copie $N_i$, et chaque arc $a$ de $N$ vers un nœud $M$, étiqueté par $\langle q, f\rangle$, on crée un arc $a'$ de $N_i$ vers $M$, étiqueté par $val(a)$ et prenant pour valuation $\langle q' \times f_i, f' \times f_i\rangle$.

Pour obtenir un SLDD$_\times$ plutôt qu'un SLDD$_+$, on « normalise » le diagramme affine de manière à assurer que toutes les valuations sont de la forme $\langle 0, f\rangle$. Pour cela, on crée autant de copies $N_1, \ldots, N_k$ de $N$ que de facteurs $q_1 + f_1$, …, $q_k + f_k$ différents sur les arcs $a_j$ entrant dans $N$. Chaque copie $a'$ d'un $a \in In(N)$ porte la valuation $\langle 0, f_a\rangle$ le facteur additif est reporté sur les arcs suivants : pour chaque copie $N_i$, et chaque arc $a$ de $N$ vers un nœud $M$, étiqueté par $\langle q, f\rangle$, on crée un arc $a'$ de $N_i$ vers $M$, étiqueté par $val(a)$ et prenant pour valuation $\langle q' + \frac{q_i}{f_i}, f'\rangle$. Finalement, les étiquettes des arcs entrant dans le puits sont mises sous la forme $\langle q + f, 0\rangle$.

Enfin, pour obtenir un ADD plutôt qu'un SLDD à partir d'un AADD, on crée autant de copies $N_i$ de $N$ que de valuations $q_i + f_i$ différentes sur les arcs entrant dans $N$ et le facteur $q_i + f_i$ est repoussé sur les arcs sortant des $N_i$ : les étiquettes $\langle q', f'\rangle$ des arcs $a'$ deviennent $\langle q_i + f_i \times q', f_i \times f'\rangle$.

Notons que ces transformations peuvent faire croître exponentiellement la taille du diagramme, ce qui est inévitable dans le pire cas : la fonction $f(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} 2^{i-1} x_i$ sur $\{0,1\}^n$ par exemple, prend $2^n$ valeurs différentes. Sa représentation par un ADD possède donc $2^n$ feuilles, alors qu'elle peut être représentée par un SLDD$_+$ (et donc par un AADD) à $n+1$ nœuds et $2n$ arcs. Formellement, le langage AADD est strictement plus succinct que celui des SLDD$_+$, lui-même strictement plus succinct que ADD. Il est également strictement plus succinct que le langage SLDD$_\times$ lui-même strictement plus succinct que ADD (Fargier *et al.*, 2013b).

Par ailleurs, le nombre d'opérations effectuées lors de la transformation en AADD et d'un AADD vers un autre langage augmente le risque d'erreurs d'arrondis (on trouve en effet plus facilement dans les AADD des additions entre deux valeurs d'ordres de grandeur différents, dont le résultat sera nécessairement arrondi, oubliant la valeur la plus faible). C'est pourquoi, dans les faits, nous évitons les traductions de AADD vers SLDD et ADD. Par exemple, pour transformer un SLDD$_+$ en SLDD$_\times$ (ou inversement), on passe par le langage ADD et non AADD. Même si la transformation du SLDD en ADD n'est en théorie pas toujours possible (à cause du risque d'explosion en espace), elle l'est souvent en pratique, et évite d'obtenir un résultat erroné.

### *4.4. Requêtes et transformations*

*OPT*

L'optimisation sur un VDD $\alpha$ est une requête qui a pour but de déterminer une instanciation $\vec{x} \in D_X$ qui optimise (minimise ou maximise) la valeur $f_\alpha(\vec{x})$. Cette opération peut être effectuée de manière générale (i.e., sur un SLDD normalisé ou pas) en temps linéaire dans la taille du graphe par un algorithme de plus court (resp. de plus long) chemin dans un graphe sans circuit (le diagramme). On parcourt simplement le diagramme en largeur d'abord, et en annotant chaque nœud atteint par la valuation optimale depuis la source vers ce nœud ; on mémorise aussi au niveau du nœud le chemin correspondant (ou l'un d'entre eux s'il en existe plusieurs). On retrouve donc sur le nœud terminal la valuation optimale ainsi qu'un des chemins optimaux. Le chemin optimal fournit une instanciation $\vec{x}$ optimale.

La normalisation d'un SLDD garantissant l'existence d'un chemin de poids neutre sortant de chaque nœud, la valeur minimale (resp. maximale) d'un SLDD$_+$ (resp. SLDD$_\times$) est donnée directement par l'offset, et l'instanciation correspondante est obtenue en suivant systématiquement un chemin de poids 0 (resp. 1). Dans le cas des AADD normalisés, l'offset $\langle q_0, f_0 \rangle$ fournit et la valuation minimale $(q_0)$ et la valuation maximale $(q_0 + f_0)$. On obtient une instanciation de valeur minimale en suivant systématiquement les arcs de valuation $q = 0$, et une instanciation de valeur maximale en suivant systématiquement les arcs de valuation $q + f = 1$.

*CD*

Le conditionnement est une transformation qui, pour un VDD $\alpha$ et une instanciation $\vec{x}$ d'un sous-ensemble de variables de $X$ renvoie un graphe représentant la

restriction de $f_\alpha$ à $\vec{x}$. Il peut être réalisé en un temps polynomial pour chacun des langages étudiés dans cet article : on supprime tout arc issu d'un nœud étiqueté par une variable instanciée par $\vec{x}$ mais portant une valeur différente de celle indiquée dans cette instanciation. On supprime ensuite les éventuels nœuds sans arc entrant, et on normalise le VDD [3].

$\otimes BC$

La transformation $\otimes BC$ associe à deux VDD $\alpha$ et $\beta$, de même nature et ordonnés selon le même ordre, un VDD $\gamma$ du même langage tel que $f_\gamma = f_\alpha \otimes f_\beta$. L'algorithme 6, dont le fonctionnement reprend le principe du produit d'automates, est un algorithme polynomial réalisant cette opération sur deux SLDD$_\otimes$, sous l'hypothèse que l'opérateur $\otimes$ est associatif. Cette algorithme crée pour le nouveau SLDD$_\otimes$ $\gamma$ un nœud $M_\gamma$ pour chaque couple de nœuds $M_\alpha$ et $M_\beta$ pris dans un niveau donné des automates d'origine ($M_\gamma$, $M_\beta$ et $M_\alpha$ partagent donc la même variable) ; on mémorise les nœuds $M_\alpha$ et $M_\beta$ dont est issu le nœud $M_\gamma$ dans les tables $link_\alpha$ et $link_\beta$. L'arc $a$ de $\gamma$ sortant de $M_\gamma$ porte la valuation $\phi(a_\gamma) = \phi(a_\alpha) \otimes \phi(a_\beta)$, où $a_\alpha$ et $a_\beta$ sont les arcs sortants des nœuds $link_\alpha(M_\gamma)$ et $link_\beta(M_\gamma)$ et tels que $val(a_\gamma) = val(a_\alpha) = val(a_\beta)$.

## 5. Résultats expérimentaux

Dans cette section, nous comparons expérimentalement, d'une part, l'efficacité des différentes heuristiques, et, d'autre part, la compacité pratique des différents types de VDD décrits dans les sections précédentes.

Nous avons testé nos structures de données et heuristiques sur deux familles de jeux d'essai. La première est un jeu d'essai standard contenant des réseaux bayésiens (Cozman, 2002). La seconde nous a été fournie par Renault et contient des VCSP additifs représentant des problèmes de configuration de voitures. Ces instances sont composées de contraintes « dures », définissant les modèles de voitures faisables ainsi que de contraintes valuées, le prix d'un véhicule étant la somme des coût spécifiés par les différentes contraintes valuées. Trois jeux d'essai nommés **Small**, **Medium** et **Big** représentent trois types différents de voitures (deux citadines et un utilitaire). Les caractéristiques de ces jeux d'essai sont les suivantes :

– **Small** : #variables=139; taille du domaine max=16; #contraintes=176 (incluant 29 contraintes de coût)

– **Medium** : #variables=148; taille du domaine max=20; #contraintes=268 (incluant 94 contraintes de coût)

– **Big** : #variables=268; taille du domaine max=324; #contraintes=2 157 (incluant 1 825 contraintes de coût)

3. On peut également ne pas supprimer ces arcs, mais seulement les rendre « invalides ». On ne modifie alors pas réellement le graphe mais les arcs invalidés sont dorénavant ignorés.

**Algorithme 6 :** $\otimes BC(\alpha, \beta)$

input : $\alpha, \beta$ deux SLDD$_\otimes$ ordonnés (même ordre) avec $\otimes$ associatif
output : Un SLDD$_\otimes\gamma$ tel que $\gamma = \alpha \otimes \beta$
// $link_\alpha$ et $link_\beta$ mémorisent pour chaque nœud créé ses nœuds d'origine dans $\alpha$ et $\beta$
// $NoeudProd(M_\alpha, M_\beta)$ désigne le nœud créé à partir des nœuds $M_\alpha, M_\beta$

$\phi_0(\gamma) \leftarrow \phi_0(\alpha) \otimes \phi_0(\beta)$; // calcul de l'offset de $\gamma$
$M_\alpha \leftarrow root(\alpha)$;
$M_\beta \leftarrow root(\beta)$;
// La source de $\gamma$ est le produit des sources de $\alpha$ et $\beta$
Créer un nouveau nœud $M_\gamma$ avec $Var(M_\gamma) = Var(M_\alpha) = Var(M_\beta)$;
$root(\gamma) \leftarrow M_\gamma$;
$link_\alpha(M_\gamma) \leftarrow M_\alpha$;
$link_\beta(M_\gamma) \leftarrow M_\beta$;
$NoeudProd(M_\alpha, M_\beta) \leftarrow M_\gamma$;
for *each* $x \in X$, do
  for *each* $M_\gamma$ *déjà créé tel que* $Var(M_\gamma) = x$ do
    for *each* $v \in D_x$ do
      Ajouter à $M_\gamma$ un arc sortant $a_\gamma$ portant la valeur $v$;
      $a_\alpha \leftarrow$ arc sortant de $link_\alpha(M_\gamma)$ tel que $val(a_\alpha) = v$;
      $a_\beta \leftarrow$ arc sortant de $link_\beta(M_\gamma)$ tel que $val(a_\beta) = v$;
      // $a_\gamma$ doit pointer sur le nœud produit des nœuds pointés par $a_\alpha$ et $a_\beta$
      if $\nexists\, M'_\gamma$ *tel que* $link_\alpha(M'_\gamma) = Out(a_\alpha)$ *et* $link_\beta(M_\gamma) = Out(a_\beta)$ then
        // $M'_\gamma$, le produit des nœuds issus de $a_\alpha$ et $a_\beta$ n'a pas encore été créé : on le crée.
        Créer un nouveau nœud $M'_\gamma$;
        $link_\alpha(M'_\gamma) \leftarrow Out(a_\alpha)$;
        $link_\beta(M_\gamma) \leftarrow Out(a_\beta)$;
        $NoeudProd(M_\alpha, M_\beta) \leftarrow M'_\gamma$;
      $Out(a_\gamma) \leftarrow Prod(M_\alpha, M_\beta)$;
      $\phi(a_\gamma) \leftarrow \phi(a_\alpha) \otimes \phi(a_\beta)$;
NormaliseSLDD$(\gamma, \phi_0(\gamma))$;

Enfin, les problèmes **Small Price Only**, **Medium Price Only** et **Big Price Only** sont constitués des seules contraintes de coût des instances **Small**, **Medium** et **Big**, respectivement (les contraintes « dures » sont omises)[4].

4. Ces jeux d'essai ainsi que le code source utilisé peuvent être téléchargés depuis les pages
http://www.irit.fr/~Helene.Fargier/BR4CP/benches.html
http://www.irit.fr/~Helene.Fargier/BR4CP/CompilateurVDD.html

### *5.1. Efficacité des heuristiques*

Pour tester l'efficacité des différentes heuristiques, nous avons mesuré, pour chacune d'elles, le temps de compilation de chaque instance en SLDD, ainsi que le nombre d'arcs du diagramme résultant. Les problèmes de configuration avec fonction de coût ont été compilés en $\text{SLDD}_+$, les instances de réseaux bayésiens ont été compilées en $\text{SLDD}_\times$. Les résultats sont présentés en tableau 1. Les notations t-o et m-o signifient respectivement *time-out* ( >1h) et *out of memory* ( >2Gio). Toutes les expérimentations ont été effectuées sur un processeur Intel i7 à 2,7GHz 4 cœurs et avec 4Gio de RAM. L'implémentation est réalisée en Java avec OpenJDK 7.

Il apparaît que MCS-Inv est généralement l'heuristique la plus performante, tant du point de vue de la taille du diagramme généré que de celui du temps de calcul. Les résultats de MCF sont mauvais lorsqu'il n'y a que des contraintes souples (problèmes **Price Only** et réseaux bayésiens). Néanmoins, les performances de MCF sont supérieures à celles de *Band-width* (au moins pour la taille) sur les problèmes alliant contraintes souples et contraintes dures, et même supérieures à celles de MCS-Inv sur le problème **Medium** en ce qui concerne la taille du SLDD obtenu. Notons également la faiblesse de *Force* en toutes circonstances. Au vu de ces résultats, nous avons choisi d'utiliser MCS-Inv pour la suite des expérimentations.

*Tableau 1. Comparaison des heuristiques MCF, Band-Width, MCS-Inv et Force*

| | MCF | | *Band-Width* | | MCS-Inv | | *Force* | |
|---|---|---|---|---|---|---|---|---|
| Instance | arcs | tps | arcs | tps | arcs | tps | arcs | tps |
| $\text{VCSP}\mapsto\text{SLDD}_+$ | | | | | | | | |
| Sml. Price Only | 239 | 0,2s | 120 | 0,3s | 108 | 0,2s | 2 946 | 0,4s |
| Med. Price Only | 2 009 | 0,5s | 906 | 0,5s | 499 | 0,4s | 40 474 | 11,6s |
| Big Price Only | m-o | - | 251 594 | 33s | 9 687 | 3,8s | - | t-o |
| Small | 7 564 | 1,1s | 10 451 | 1,0s | 5 584 | 0,8s | 9 198 | 0,8s |
| Medium | 19 363 | 1,8s | 30 835 | 1,4s | 17 062 | 1,3s | 34 263 | 1,5s |
| Big | m-o | - | 1 339 821 | 139s | 921 666 | 80s | m-o | - |
| $\text{Bayes}\mapsto\text{SLDD}_\times$ | | | | | | | | |
| Cancer | 25 | 0,1s | 25 | 0,09s | 25 | 0,04s | 25 | 0,05s |
| Asia | 71 | 0,08s | 57 | 0,09s | 45 | 0,05s | 49 | 0,07s |
| Car-starts | 157 | 0,2s | 83 | 0,1s | 83 | 0,1s | 99 | 0,1s |
| Alarm | m-o | - | 14899 | 0,5s | 3993 | 0,5s | 20134 | 1,2s |

### *5.2. Efficacité spatiale des VDD*

Nous avons voulu comparer la taille des diagrammes obtenus pour les différents langages considérés dans l'article. Les jeux d'essai ont été compilés sous la forme de SLDD ($\text{SLDD}_+$ pour les VCSP et $\text{SLDD}_\times$ pour les réseaux bayésiens) puis traduits dans les autres langages. Les tableaux 2 et 3 indiquent les tailles obtenues pour la représentation des problèmes de configuration et des réseaux bayésiens sous la forme de ADD, $\text{SLDD}_+$, $\text{SLDD}_\times$ et AADD, ainsi que les temps de compilation en $\text{SLDD}_+$. Le temps consommé par les transformations n'est pas une information importante ici car il n'est pas lié au temps de compilation initial, et il ne dépend que de la taille maximale atteinte au cours de la transformation.

*Tableau 2. Compilation de problèmes de configuration en SLDD$_+$ et transformations en ADD, SLDD$_\times$ et AADD*

| VCSP | SLDD$_+$ | | ADD | SLDD$_\times$ | AADD |
|---|---|---|---|---|---|
| Instance | arcs | temps | arcs | arcs | arcs |
| Small Price only | 108 | 0,2s | 7 439 | 7 439 | 108 |
| Medium Price only | 499 | 0,4s | 99 280 | 99 280 | 495 |
| Big Price only | 9 687 | 3,8s | m-o | - | 9 687 |
| Small | 5 584 | 0,8s | 637 319 | 33 639 | 5 584 |
| Medium | 17 062 | 1,3s | 2 071 474 | 314 648 | 17 062 |
| Big | 921 666 | 80s | m-o | - | 921 666 |

*Tableau 3. Compilation de réseaux bayésiens en SLDD$_\times$ et transformations en ADD, SLDD$_+$ et AADD*

| Rés. bay. | SLDD$_\times$ | | ADD | SLDD$_+$ | AADD |
|---|---|---|---|---|---|
| Instance | arcs | temps | arcs | arcs | arcs |
| Cancer | 25 | $0,04s$ | 45 | 45 | 21 |
| Asia | 45 | $0,05s$ | 431 | 431 | 45 |
| Car-starts | 83 | $0,1s$ | 64 029 | 39 265 | 77 |
| Alarm | 3 993 | $0,5s$ | m-o | - | 3 993 |

Ces expérimentations confirment en pratique les résultats théoriques de compacité des différents langages (Fargier *et al.*, 2013b). En effet, on retrouve bien en pratique que le langage ADD (langage le moins succinct) est toujours moins compact que SLDD$_+$, SLDD$_\times$ et AADD. À l'inverse, AADD est toujours au moins aussi compact que SLDD$_+$, SLDD$_\times$ et ADD. Il s'avère qu'un langage offre en pratique une bonne compacité si celui-ci intègre l'opérateur adéquat au type d'instances considéré. Ainsi, les langages SLDD$_+$ et AADD qui intègrent l'addition sont plus efficaces spatialement pour la compilation de CSP pondérés dont les contraintes sont de nature additive (portant sur un prix), alors que SLDD$_\times$ et AADD sont plus compacts pour la compilation de réseaux bayésiens, où les contraintes sont de nature multiplicative (ce sont des tables de probabilités conditionnelles). À l'inverse, un opérateur non pertinent n'apporte que peu, voire pas, d'amélioration. Ainsi, la comparaison entre les différents langages de type SLDD et le langage AADD nous montre que l'utilisation d'un deuxième opérateur n'apporte pas, dans les problèmes purement additifs ou purement multiplicatifs, d'amélioration pratique en terme de compacité. Autrement dit, l'utilisation d'un AADD par rapport à un SLDD$_+$ (resp. SLDD$_\times$) n'apporte pas de réelle amélioration pour la compilation des problèmes de configuration avec coût (resp. de réseaux bayésiens) [5].

L'utilisation du langage AADD peut également engendrer, de par une procédure de normalisation complexe, des erreurs d'arrondis, gênant notamment le hachage des nœuds lors de la détection des graphes isomorphes. En pratique, la compilation en

5. Lors de la compilation de réseaux bayésiens en ADD et AADD, nous obtenons des représentations nettement moins succinctes que celles décrites dans (Sanner, McAllester, 2005). Ceci s'explique par le choix que nous avons fait de représenter les valeurs réelles avec une précision plus importante. Dans le cas des ADD, le nombre de valeurs finales possible explose clairement quand la précision augmente.

SLDD$_\times$ n'est que peu sujette aux erreurs d'arrondis, et la compilation en SLDD$_+$ n'en souffre pas du tout. L'utilisation de valeurs exactes, sous forme de fraction, dans la compilation de SLDD$_\times$ nous a permis de vérifier qu'il n'y avait aucune erreur de normalisation due aux approximations dans ces exemples. Cependant, la taille de ces structures (des fractions) devient très rapidement trop importante pour que les structures puissent être codées avec des types primitifs (ici le type *long*, codé sur 8 octets), et l'utilisation de types sans limitation de taille multiplie par 10 en moyenne les temps de calcul.

## 6. Conclusion

Dans cet article nous avons étudié la compilation vers et la traduction entre différents types de diagrammes de décision valués. Du point de vue théorique le langage AADD peut être vu comme une généralisation des langages SLDD$_+$ et SLDD$_\times$, langages qui généralisent eux-mêmes le langage ADD : AADD est, parmi eux, le langage de compilation le plus succinct. Les expérimentations montrent que, dans la pratique, pour la compilation de problèmes de nature purement additive (resp. purement multiplicative), seul l'opérateur $+$ (respectivement $\times$) est utile, et que l'utilisation de SLDD$_+$ (resp. SLDD$_\times$) suffit amplement à la compilation du problème original. Nous avons également étudié plusieurs heuristiques, et vérifié que les meilleures tendaient au regroupement des variables incluses dans un même groupe de contraintes. Pour ce qui est des travaux futurs, nous planifions d'implémenter plusieurs requêtes et transformations opérant sur des VDD, afin de pouvoir utiliser ces structures pour traiter efficacement (et avec une garantie de temps de réponse) des problèmes de configuration en ligne avec maintien d'un indicateur de coût minimal.

*Remerciements*

*Cet article étend les résultats préliminaires présentés dans (Fargier* et al.*, 2013a). Les auteurs souhaitent remercier les relecteurs de l'article pour leurs commentaires. Le travail réalisé a bénéficié du support du projet BR4CP ANR-11-BS02-008 de l'Agence Nationale de la Recherche.*

## Bibliographie

Aloul F. A., Markov I. L., Sakallah K. A. (2003). Force: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the ACM Great Lakes symposium on VLSI (GLSVLSI)*, p. 116-119.

Amilhastre J. (1999). Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes. *Thèse de doctorat, Université Montpellier II*.

Amilhastre J., Fargier H., Marquis P. (2002). Consistency restoration and explanations in dynamic CSPs application to configuration. *Artificial Intellligence*, vol. 135, p. 199-234.

Bahar R. I., Frohm E. A., Gaona C. M., Hachtel G. D., Macii E., Pardo A. et al. (1993). Algebraic decision diagrams and their applications. In *Proceedings of the International Conference Computer-Aided Design (ICCAD)*, p. 188-191.

Bistarelli S., Montanari U., Rossi F., Schiex T., Verfaillie G., Fargier H. (1999). Semiring-Based CSPs and Valued CSPs: Frameworks, Properties, and Comparison. *Constraints*, vol. 4, n° 3, p. 199-240.

Bryant R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, vol. 35, p. 677–691.

Cozman F. G. (2002). *JavaBayes Version 0.347, Bayesian Networks in Java, User Manual*. Rapport technique. (Benchmarks at http://sites.poli.usp.br/pmr/ltd/Software/javabayes/)

Drechsler R. (2002). Evaluation of static variable ordering heuristics for mdd construction. In *Proceedings of the 32nd International Symposium on Multiple-Valued Logic (ISMVL)*, p. 254-260.

Fargier H., Marquis P., Schmidt N. (2013a). Compacité pratique des diagrammes de décision valués : normalisation, heuristiques et expérimentations. In *9ème Journées Francophones de Programmation par Contraintes (JFPC)*. Aix-en-Provence. (actes électroniques)

Fargier H., Marquis P., Schmidt N. (2013b). Semiring labelled decision diagrams, revisited: Canonicity and spatial efficiency issues. In *Proceedings of the Twenty-third Joint Conference on Artificial Intelligence (IJCAI)*, p. 884-890.

Hadzic T., Andersen H. R. (2006). A BDD-based polytime algorithm for cost-bounded interactive configuration. In *Proceedings of the 21st AAAI Conference on Artificial Intelligence (AAAI)*, p. 62-67.

Hoey J., St-Aubin R., Hu A., Boutilier C. (1999). SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, p. 279–288. San Francisco, CA, Morgan Kaufmann.

Lai Y.-T., Pedram M., Vrudhula S. B. K. (1996). Formal verification using edge-valued binary decision diagrams. *IEEE Transactions on Computers*, vol. 45, n° 2, p. 247-255.

Lai Y.-T., Sastry S. (1992). Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC)*, p. 608-613.

Roussel O., Lecoutre C. (2009, feb). *XML Representation of Constraint Networks: Format XCSP 2.1*. Rapport technique. Computing Research Repository (CoRR) abs/0902.2362.

Sanner S., McAllester D. A. (2005). Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, p. 1384-1390.

Tafertshofer P., Pedram M. (1997). Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, vol. 10, n° 2-3.

Tarjan R. E., Yannakakis M. (1984, juillet). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, vol. 13, n° 3, p. 566–579.

Wilson N. (2005). Decision diagrams for the computation of semiring valuations. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, p. 331-336.