UNIVERSITÉ D'ARTOIS

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ D'ARTOIS

Spécialité : *Intelligence Artificielle*

Présentée et soutenue le *17/09/2015* **par :**

Nicolas Schmidt

Compilation de préférences
application à la configuration de produit

Directeurs de Thèse

Pierre MARQUIS
Professeur
CRIL, Université d'Artois

Hélène FARGIER
Directrice de recherche CNRS
IRIT, Université Paul Sabatier

Rapporteurs

Bruno ZANUTTINI
Maître de conférences HDR
GREYC, Université de Caen

Laurent SIMON
Professeur
LaBRI, Université Bordeaux 1

Examinateurs

Daniel LE BERRE
Professeur
CRIL, Université d'Artois

Élise VAREILLES
Maître-assistant
Ecole des Mines d'Albi

Unités de Recherche :
CRIL - IRIT/ADRIA
École doctorale :
Ecole Doctorale Sciences Pour l'Ingénieur

Résumé

L'intérêt des différents langages de la famille des diagrammes de décision valués (VDD) est qu'ils admettent des algorithmes en temps polynomial pour des traitements (comme l'optimisation, la cohérence inverse globale, l'inférence) qui ne sont pas polynomiaux (sous l'hypothèse $P \neq NP$), si ils sont effectués sur le problème dans sa forme originale tel que les réseaux de contraintes ou les réseaux bayésiens.

Dans cette thèse, nous nous intéressons au problème de configuration de produit, et plus spécifiquement, la configuration en ligne avec fonction de valuation associée (typiquement, un prix). Ici, la présence d'un utilisateur en ligne nous impose une réponse rapide à ses requêtes, rapidité rendant impossible l'utilisation de langages n'admettant pas d'algorithmes en temps polynomial pour ces requêtes. La solution proposée est de compiler hors-ligne ces problèmes vers des langages satisfaisant ces requêtes, afin de diminuer le temps de réponse pour l'utilisateur.

Une première partie de cette thèse est consacrée à l'étude théorique des VDD, et plus particulièrement les trois langages *Algebraic Decision Diagrams*, *Semi ring Labelled Decision Diagrams* et *Affine Algebraic Decision Diagrams* (ADD, SLDD et AADD). Nous y remanions le cadre de définition des SLDD, proposons des procédures de traductions entre ces langages, et étudions la compacité théorique de ces langages. Nous établissons dans une deuxième partie la carte de compilation de ces langages, dans laquelle nous déterminons la complexité algorithmique d'un ensemble de requêtes et transformations correspondant à nos besoins. Nous proposons également un algorithme de compilation à approche ascendante, ainsi que plusieurs heuristiques d'ordonnancement de variables et contraintes visant à minimiser la taille de la représentation après compilation ainsi que le temps de compilation. Enfin la dernière partie est consacrée à l'étude expérimentale de la compilation et de l'utilisation de formes compilées pour la configuration de produit. Ces expérimentations confirment l'intérêt de notre approche pour la configuration en ligne de produit.

Nous avons implémenté au cours de cette thèse un compilateur (le compilateur SALADD) pleinement fonctionnel, réalisant la compilation de réseaux de contraintes et de réseaux bayésiens, et avons développés un ensemble de fonctions adaptées à la configuration de produit. Le bon fonctionnement et les bonnes performances de ce compilateur ont été validés via un protocole de validation commun à plusieurs solveurs.

Mots-clefs : Compilation, configuration de produit, recommandation, diagramme de décision valué, heuristique d'ordonnancement, CSP

Abstract

The different languages from the valued decision diagrams (VDD) family benefit from polynomial-time algorithms for some tasks of interest (such as optimization, global inverse consistency, inference) for which no polynomial-time algorithm exists (unless $P = NP$) when the input is a constraint network or a Bayesian network considered at start.

In this thesis, we focus on configuration product problems, and more specifically on-line configuration with an associated valuation function (typically, a price). In this case, the existence of an on-line user forces us to quickly answer to his requests, making impossible the use of languages that does not admit polynomial-time algorithm for this requests. Therefore, our solution consists in an off-line compilation of these problems into languages that admit such polynomial-time algorithms, and thus decreasing the latency for the user.

The first part of this thesis is dedicated to the theoretical study of VDDs, an more specifically Algebraic Decision Diagrams (ADDs), Semi ring Labelled Decision Diagrams (SLDDs) and Affine Algebraic Decision Diagrams (AADDs). We revisit the SLDD framework, propose translation procedures between these languages and study the succinctness of these languages. In a second part, we establish a knowledge compilation map of these languages, in which we determine the complexity of requests and transformations corresponding to our needs. We also propose a bottom-up compilation algorithm and several variables and constraints ordering heuristics whose aim is to reduce the size of the compiled form, and the compilation time. The last part is an experimental study of the compilation and the use of the compiled form in product configuration. These experimentations confirm the interest of our approach for on-line product configuration.

We also implemented a fully functional bottom-up compiler (the SALADD compiler), which is capable of compiling constraints network and Bayesian network into SLDDs. We also developed a set of functions dedicated to product configuration. The proper functioning and good performances of this program was validated by a validation protocol common to several solvers.

Key-words: Compilation, product configuration, recommendation, valued decision diagram, ordering heuristic, CSP

Table des matières

Chapitre 1

Introduction

Dans les approches interactives de résolution de problèmes contraints en ligne, c'est l'utilisateur, et non la machine, qui résout un problème combinatoire d'optimisation de préférences. Les problèmes de configuration de produit en sont des exemples typiques (on configure interactivement une voiture, un ordinateur, etc). En laissant le client explorer l'offre, la configuration en ligne doit lui permettre de maximiser sa satisfaction. Dans de nombreux domaine, les fabricants proposent à leurs utilisateurs un nombre croissant d'options et de paramètres, si bien que Renault par exemple se vante depuis bien longtemps que tous les modèles et configurations possibles de voitures mis bout à bout représenteraient la distance Terre-Soleil. C'est bien sûr faux... On sortirait laaargement du système solaire.

S'il n'est pas envisageable de répertorier toutes les combinaisons possibles dans un catalogue, il n'est généralement pas non plus possible de laisser l'utilisateur configurer librement son produit, ce derniers répondant à un ensemble de contraintes venant du fabriquant et pouvant être autant techniques que commerciales. L'un des principaux freins à la vente en ligne de produits configurables est la difficulté qu'a l'internaute à se diriger sur un produit viable qui satisfait ses préférences, et de manière orthogonale la difficulté qu'a l'e-commerçant à orienter son client, difficulté directement liée à la dimension combinatoire de l'e-catalogue.

Différentes fonctionnalités peuvent aider l'utilisateur dans sa configuration, que ce soit en lui indiquant les choix qui ne peuvent aboutir à une solution viable, en lui donnant une fourchette de prix correspondant à sa configuration en cours, en indiquant les choix permettant de minimiser le coût final du produit, les choix qu'il lui reste à faire, ou encore les choix effectués sur d'autres paramètres/options qui le bloquent dans sa recherche courante. Ces fonctionnalités correspondent à des requêtes qui sont fortement combinatoires

(NP-difficiles). Lorsque le nombre de paramètres configurables devient important, il est alors compliqué, voire impossible, de répondre à ces requêtes en un temps satisfaisant pour l'utilisateur d'un configurateur en ligne.

D'où l'idée de prétraiter, de « compiler », le problème original sous une forme qui permet un traitement efficace des requêtes. Cette compilation s'effectue hors ligne, avant la phase de requêtes, et n'a besoin d'être effectuée qu'une seule fois.

Selon les formes compilées, ces requêtes peuvent alors être calculées en un temps linéaire, polynomial ou exponentiel dans la taille du problème compilé. Un compromis entre l'efficacité et la compacité de la forme compilée est donc à trouver.

Dans cette thèse, nous étudions une famille de ces « langages » de compilation : les VDD (pour *Valued Decision Diagrams*). Les diagrammes de décision, et notamment les bien connus OBDD (*Ordered Binary Decision Diagram*), ont depuis longtemps montré leurs utilité en tant que langage de représentation et de traitement de données, dans de nombreux domaines, notamment dans des problèmes de prises de décision, de diagnostic ou de vérification formelle de modèles. La dimension « valuée » permet d'ajouter à la représentation une notion, par exemple, de prix, de probabilité, ou encore d'utilité d'une solution.

Dans la suite, ces langages sont étudiés autant théoriquement qu'expérimentalement, par le biais d'un compilateur implémenté par nos soins. Il nous permet d'étudier la compacité pratique et les performances des différents langages de représentation considérés, ainsi que de tester les algorithmes proposés et de prouver la faisabilité de nos méthodes de compilation.

Cette thèse est divisée en trois parties. La première, théorique, est consacrée aux langages de représentation eux-mêmes. En plus d'apporter un état de l'art, et de mettre en place un certain nombres de notations, cette partie se focalise sur les principaux langages étudiés, tels qu'ils ont été introduits, et tels que nous les utilisons. La deuxième partie est consacrée à la compilation et à l'exploitation des données compilées. Nous y présentons des résultats sur la complexité de requêtes pour chacun des langages, nos algorithmes, ainsi que des heuristiques d'ordonnancement de variables et de contraintes. La troisième partie, expérimentale, présente divers résultats obtenu par notre compilateurs sur plusieurs jeux d'essai, ainsi qu'une analyse « pratique » des différents langages et méthodes. Nous testons notre compilateur notamment au travers d'un protocole, et étudions d'autres applications possibles.

Première partie

Contexte et langages de représentation

Chapitre 2

Contexte

Il est ici principalement question de format dans lequel sont exprimées les données. De même que le plan de montage d'un meuble suédois est plus rapide à visualiser et à interpréter que sa notice textuelle de montage (en suédois ?), tous les formats ou langages, dans lesquels peut être décrit un problème ne se valent pas. De la façon dont il est décrit dépendent grandement la difficulté et la rapidité de résolution d'un problème.

Certains langages tels que les réseaux de contraintes [Montanari, 1974] sont parfaitement adaptés à l'écriture des entrelacs de contraintes qui interviennent en configuration de produit. Cependant, même si ces langages sont performants pour la résolution automatique de problèmes, la propagation de contraintes ou la recherche de solutions par la machine, ils sont moins adaptés à la résolution de problèmes liés à la configuration en ligne.

En effet, la requête principale que l'on demande de traiter à un configurateur de produit en ligne est « l'ensemble de choix que je viens de faire peut-il conduire à une configuration réalisable ? », autrement dit le maintien de la cohérence globale du problème. Cette requête est NP-difficile lorsque le problème est exprimé sous la forme d'un réseau de contraintes.

Cela ne rend pas les réseaux de contraintes inutilisables pour la configuration de produit en ligne, où cette requête doit être exécuté en un temps très cours afin de garantir une fluidité de navigation pour l'utilisateur. Par exemple Bessiere *et al.* [2013] proposent un algorithme calculant la cohérence globale inverse, ou GIC* (*Global Inverse Consistency*) permettant ainsi de ne pas proposer à un utilisateur les choix ne conduisant pas à une solution. Cet algorithme semble donner de bons résultats en termes de temps, même sur des problèmes complexes, à condition qu'il ne s'agisse que d'une mise à jour de l'information, après un changement mineur (l'affectation d'une variable). Cependant cet algorithme n'offre aucune garantie de performance (temporelle).

De plus la cohérence n'est pas la seule requête qui nous intéresse ici. En effet, dans cette thèse, nous travaillons avec des problèmes valués. Ainsi toutes les solutions ne se valent pas, et à chaque affectation de l'ensemble des variables est associé une valuation pouvant correspondre à un prix, une utilité, une probabilité, etc.

Il peut être intéressant de donner à l'utilisateur des retours concernant par exemple le prix minimal que peut avoir son produit en cours de configuration, ou quel choix parmi plusieurs conduira au produit le moins cher. Ces requêtes d'optimisation de la valuation finale correspondent également à des problèmes NP-difficiles lorsque les données sont exprimées dans le langage des réseaux de contraintes.

Des langages conduisant à une complexité plus faible (polynomiale voire même linéaire dans la taille de la représentation) de ces requêtes existent. Leur utilisation nécessite cependant une étape de compilation. Cette compilation est une étape complexe, et dont la faisabilité n'est pas garantie (les limites pouvant être tant spatiales que temporelles). Cependant, une fois cette compilation réalisée, la complexité de ces requêtes devient plus faible, les temps de calculs sont généralement plus courts, et la configuration en ligne en est facilitée.

Le choix du langage que le configurateur utilise est donc primordial. Il nous faut adopter un langage supportant les requêtes que l'on souhaite implémenter sur notre configurateur, et le plus compact possible et ce autant pour limiter les temps de calculs de ces requêtes (directement liés à la taille des représentations obtenues) que pour assurer la facilité, voire la possibilité, de compilation de ces problèmes.

À cet effet, nous présentons dans ce chapitre tout d'abord un ensemble de langages de compilation booléens, et nous nous aidons de la précieuse carte de compilation de Darwiche et Marquis [2002] pour comparer ces différents langages en termes d'efficacité spatiale ainsi que de requêtes et transformations réalisables sur chacun d'entre eux. Forts de notre savoir sur les langages booléens, très largement explorés dans la littérature, nous considérons ensuite les langages valués afin de déterminer ceux que nous approfondirons, développerons, testerons...

Dans cette partie, en espérant rendre la lecture plus agréable, nous avons choisi de décrire les langages et concepts de façon intuitive. Il n'y a donc pas de définition formelle. Vous pourrez cependant trouver en annexe (section A) les définitions formelles de l'ensemble des notions et langages marqués d'un *.

De plus, cette partie n'a vocation qu'à donner une vue d'ensemble des langages et méthodes de compilation existantes. Les principaux langages que nous avons étudiés au cours de cette thèse sont eux définis de façon plus complète au chapitre suivant.

2.1 Compilation booléenne

Cette partie est consacrée à l'étude des langages et des méthodes de compilation existantes. Nous considérons dans un premier temps la compilation avec des langages ayant un domaine d'arrivée booléen. Ces langages ayant été maintes fois explorés, c'est d'eux que nous tirerons notre inspiration.

On exprime dans un langage L, α qui représente une fonction f au domaine d'arrivée booléen.

Comme le disent Darwiche et Marquis [2002], trois critères sont considérés comme clé dans la compilation de connaissances : la compacité du langage cible choisi, les requêtes pouvant être calculées en un temps polynomial dans la taille de la représentation, et les transformations pouvant être effectuées en un temps polynomial dans la taille de la représentation. C'est pourquoi l'étude des différents langages de représentation est un facteur clé dans la compilation. Nous allons donc dans les paragraphes qui suivent faire un rapide tour des différents langages et de leurs propriétés, en mettant l'accent sur la compacité* de chacun.

La compacité d'un langage est relative à un autre. Un langage $L1$ est dit au moins aussi succinct (ou compact) qu'un langage $L2$, noté $L1 \leqslant L2$, si pour toute fonction f, la taille de tout α représentant f dans $L1$ est inférieure ou égale, à un polynôme près, à la taille d'au moins un β représentant f dans $L2$. Un langage de représentation $L1$ est dit plus succinct qu'un langage $L2$, noté $L1 < L2$, si on à $L1 \leqslant L2$ et $L2 \nleqslant L1$. Deux langages $L1$ et $L2$ sont dit également succincts, noté $L1 \sim L2$ ssi $L1 \leqslant L2$ et $L2 \leqslant L1$.

En d'autre termes, si un langage $L1$ est au moins aussi succinct qu'un langage $L2$, cela ne veut pas nécessairement dire que la taille α sera plus petite que celle de β, mais qu'il existe un β pour lequel il n'y aura pas d'explosion en espace lors de la transformation de β en α.

Tous les langages qui vont suivre appartiennent à la famille des NNF (« formules » propositionelles sous forme normale négative). Le principe consiste à ajouter un certain nombre de restrictions à la forme de base (la NNF simple). Chaque restriction permet d'augmenter le nombre de requêtes (et parfois de transformations) réalisables en temps polynomial, mais a pour effet de diminuer la compacité du langage.

2.1.1 Langages

NNF*

Une « formule » propositionelle est sous la forme normale négative [Barwise, 1977] si ne sont utilisés que les deux seuls opérateurs « $\wedge$ » et « $\vee$ » (conjonction et disjonction), et que l'opérateur de la négation « $\neg$ » n'est utilisé que directement sur les variables.

Les « formules » NNF sont en fait des circuits, i.e. des DAG (graphe orienté acyclique), comportant une unique racine, où les nœuds terminaux sont étiquetés soit par un littéral, soit par « $\top$ » ou « $\bot$ », et les nœuds non terminaux sont étiquetés par l'opérateur de conjonction « $\wedge$ » ou de disjonction « $\vee$ », opérateurs non nécessairement binaires. La taille d'une formule f sous la forme NNF, notée $|f|$ correspond au nombre de nœuds et au nombres d'arcs. Un exemple de NNF est donné à la figure 2.1.

Cette représentation sous forme de DAG est celle que nous utiliserons jusqu'à arriver au chapitre sur les diagrammes de décision.

Toute formule propositionnelle construite à partir des connecteurs « $\wedge$ », « $\vee$ » et « $\neg$ » peut être linéairement traduite sous sa forme NNF en utilisant les équivalences suivantes :

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

Le langage NNF est un langage de représentation très succinct car aucune restriction additionnelle sur le langage n'est imposée.

d-NNF*/DNNF*/d-DNNF*

- Déterminisme* [Darwiche, 2001b] : Un NNF est dit déterministe ssi, pour chaque disjonction, les sous-formules d'une même disjonction sont nécessairement contradictoires. Autrement dit, au maximum une seule des sous-formules d'une disjonction peut être vraie en même temps (voir figure 2.2).
- Décomposabilité* [Darwiche, 2001a] : Un NNF est dit décomposable ssi, pour chaque conjonction, les sous-formules d'une même conjonction ne partagent aucune variable. Autrement dit, l'intersection des variables d'un couple de sous-formules d'une conjonction est nécessairement vide (voir figure 2.2).

Figure 2.1 – *Exemple d'une formule NNF sur trois variables x1, x2 et x3*

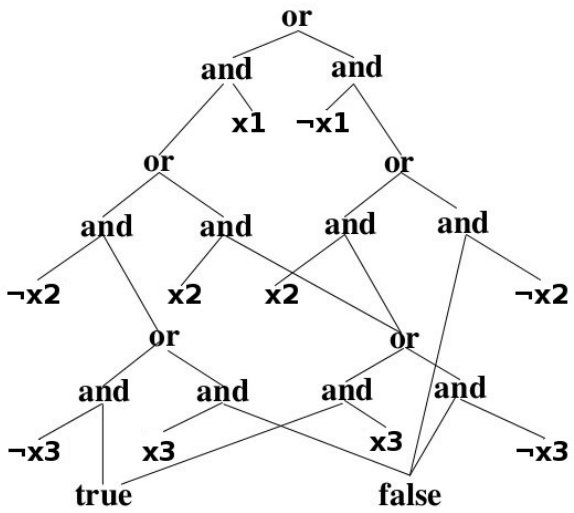
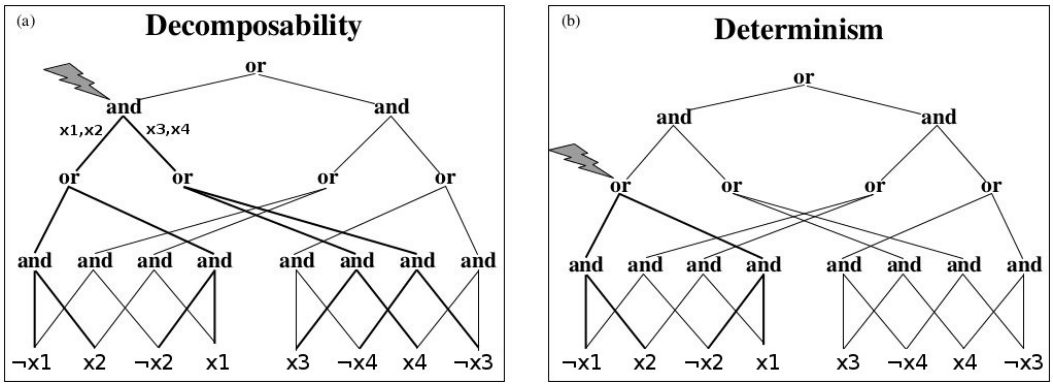


Figure 2.2 – *Une formule d-DNNF sur quatre variables x1, x2, x3 et x4. L'éclair montre en (a) un nœud ∧ décomposable, et en (b) un nœud ∨ déterministe*

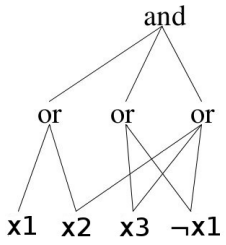


Ces deux propriétés nous conduisent non pas à un, ou à deux, mais à trois nouveaux langages : les d-NNF qui sont des NNF déterministes, les DNNF qui sont des NNF décomposables, et les d-DNNF qui, contrairement à ce que l'on peut supposer, sont exactement ce que leur nom indique (décomposables et déterministes).

Du point de vue de la compacité, nous avons logiquement NNF $\leqslant$ DNNF $\leqslant$ d-DNNF et NNF $\leqslant$ d-NNF $\leqslant$ d-DNNF, les DNNF et d-NNF étant des NNF particuliers, et les d-DNNF étant des DNNF et de d-NNF particuliers.

La réciproque étant fausse [Darwiche et Marquis, 2002] on a donc la relation de compacité NNF $<$ DNNF $<$ d-DNNF et NNF $<$ d-NNF $<$ d-DNNF.

Figure 2.3 – *La CNF* $(x1 \vee x2) \wedge (\neg x1 \vee x3) \wedge (\neg x1 \vee x2 \vee x3)$



Il existe également d'autres propriétés tel que l'aplatissement* (*flatness*), donnant les f-NNF et désignant les NNF de hauteur maximale égale à deux, ou l'uniformité* (*smoothness*), donnant les s-NNF, mais là il faudra vous rendre en annexe pour satisfaire votre curiosité.

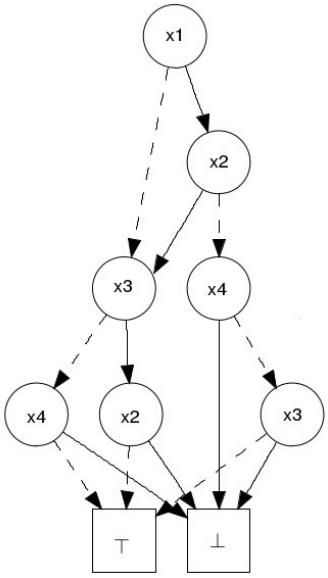
CNF/DNF

- Conjonction de littéraux : une conjonction de littéraux est un nœud étiqueté par l'opérateur « $\wedge$ » et dont les nœuds successeurs sont tous des nœuds terminaux étiquetés par un littéral, chaque variable ne pouvant apparaître qu'une seule fois dans la conjonction. Une conjonction de littéraux est aussi appelée terme ou cube.
- Disjonction de littéraux : une disjonction de littéraux est un nœud étiqueté par l'opérateur « $\vee$ » et dont les nœuds successeurs sont tous des nœuds terminaux étiquetés par un littéral, chaque variable ne pouvant apparaître qu'une seule fois dans la disjonction. Une disjonction de littéraux est aussi appelée une clause.

- Une DNF (pour formule sous forme normale disjonctive) est une disjonction de termes, c'est-à-dire une disjonction de conjonctions de littéraux. Il s'agit donc d'une f-NNF (deux arcs au maximum de la racine aux feuilles). La racine est étiquetée par l'opérateur « $\vee$ » (disjonction) et l'étage suivant n'est composé que de conjonctions de littéraux.
- Un CNF (pour formule sous forme normale conjonctive) est une conjonction de de clauses, c'est-à-dire une conjonction de disjonctions de littéraux. Il s'agit donc d'une f-NNF, la racine est nécessairement étiquetée par l'opérateur « $\wedge$ » (conjonction) et l'étage suivant n'est composé que de disjonctions de littéraux (exemple figure 2.3).

Le langage des CNF est celui utilisé notamment par les solveurs SAT, programmes permettant de déterminer la cohérence d'une formule propositionnelle.

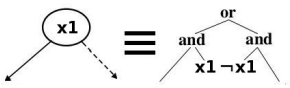
Figure 2.4 – *BDD représentant la fonction* $(\neg x3 \wedge \neg x4) \vee (\neg x1 \wedge \neg x2 \wedge x3)$



BDD*/FBDD*/OBDD*/OBDD$_{\leq}$*

Un diagramme de décision binaire (BDD) [Lee, 1959; Akers, 1978] est un DAG ne possédant qu'une seule racine, et deux nœuds terminaux. Les nœuds non terminaux sont des nœuds de décision étiquetés par une variable, et chacun des arcs sortant correspond à une affectation possible de cette variable. Les nœuds terminaux sont étiquetés par les valeurs « $\top$ » et « $\bot$ » (exemple figure 2.4).

Le langage BDD, bien que généralement considéré comme un langage à part entière, est un sous-langage de la famille des NNF. Un BDD peut être écrit sous sa forme NNF en remplaçant chaque nœud de décision par un ensemble de nœuds « type NNF » de la façon suivante :



C'est cependant bien l'écriture sous forme BDD que nous utiliserons par la suite, cette dernière étant plus lisible.

Il existe d'autres sous-langages de BDD :

- Les FBDD (*free binary decision diagram*) [Gergov et Meinel, 1994] sont des BDD satisfaisant la propriété de lecture unique (*read-once*). C'est-à-dire que quel que soit le chemin de la racine à un nœud terminal,

chaque variable ne peut être rencontrée au maximum qu'une seule fois. Cela équivaut à rajouter la restriction de décomposabilité au BDD.
- Les OBDD (*ordered binary decision diagram*) [Bryant, 1986] sont des FBDD dans lesquels les variables sont ordonnées. C'est-à-dire que quel que soit le chemin de la racine à un nœud terminal, les variables seront toujours rencontrées dans le même ordre.
- Un $\text{OBDD}_{\leq}$ est un OBDD ayant « $\leq$ » pour ordre de variables. Le sous-langage OBDD est l'union des $\text{OBDD}_{\leq}$ pour tous les ordres « $\leq$ » possibles.

Les OBDD sont beaucoup utilisés pour leur simplicité d'interprétation dans des problèmes de prise de décision, et la propriété d'unicité de la représentation (forme canonique) d'un $\text{OBDD}_{\leq}$ en fait un outil idéal pour la vérification formelle de modèles.

Faisons une petite pause dans l'énumération des langages cibles possibles afin de comparer ceux que nous avons déjà sous la main.

2.1.2 Carte de compilation

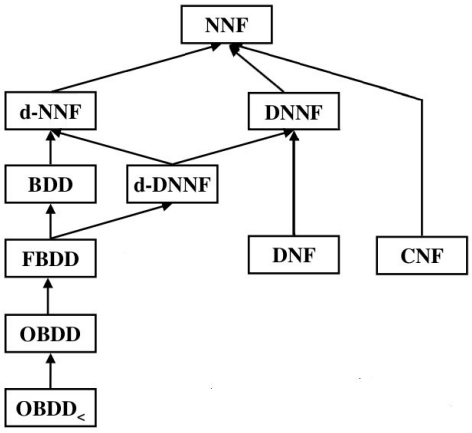
La carte de compilation a été introduite par Darwiche et Marquis [2002]. Elle vise à comparer les langages cibles pour la compilation en termes de compacité, ainsi qu'à analyser les requêtes et transformations réalisables pour chacun de ces langages. L'idée sous-jacente étant de guider le développeur dans le choix du langage de représentation de données, le choix idéal étant le langage le plus succinct possible satisfaisant les requêtes et transformations dont le développeur a besoin.

Les langages sont évalués selon trois critères : la compacité, les requêtes calculables en temps polynomial, et les transformations réalisables en temps polynomial.

Compacité théorique

La compacité d'un langage $L1$ étant exprimé par rapport à un autre langage $L2$, la compacité d'un langage n'est que relative à un autre. La figure 2.5 donne un comparatif de la compacité des différents langages étudiés précédemment les uns par rapport aux autres.

Figure 2.5 – *Comparatif de compacité des langages. $L1 \rightarrow L2$ signifie que $L2$ est strictement plus succinct que $L1$. Une absence de flèche signifie que les deux langages ne sont pas comparables en terme de compacité.*



Gardons en mémoire que cette compacité n'est « que théorique », et définie « à un polynôme près » : ainsi on peut parfaitement avoir $L1 \leqslant L2$, $\alpha \in L1$, $\beta \in$ Ł2, $\alpha \equiv \beta$ et cependant $|\alpha| > |\beta|$. La compacité est cependant une garantie de non explosion en taille.

Toute restriction supplémentaire sur un langage ne pouvant logiquement que « casser » la représentation de taille minimale qu'une formule peut avoir dans ce langage, et en aucun cas en créer une nouvelle, si un langage $L2$ est un sous ensemble du langage $L1$ (langage $L1$ auquel on aurait ajouté une restriction), alors on a $L1 \leqslant L2$. NNF, non restreint, est logiquement le langage le plus succinct parmis ceux considérés ici. Le BDD, de par la construction de chacun des nœuds de décision, est déterministe (une variable ne pouvant être vraie et fausse à la fois), on a donc d-NNF $\leqslant$ BDD. De par sa définition, FBDD est un BDD décomposable, d'où DNNF $\leqslant$ FBDD, et d-DNNF $\leqslant$ FBDD.Enfin, DNF est un sous ensemble de DNNF [Darwiche, 1999, 2001a], ce qui nous donne DNNF $\leqslant$ DNF.

Requêtes et transformations

Mais si le langage NNF est le langage le plus succinct et n'est pas restreint, quel est l'intérêt des autres langages ? Eh bien, ce sont ces restrictions sur les langages de représentation qui permettent à des algorithmes de calcul de requêtes ou de transformations, polynomiaux dans la taille de la représentation, d'exister (si un tel algorithme existe, on dit que le langage L satisfait la requête).

Par exemple, il peut être difficile de décider la cohérence (**CO**) d'une NNF. Cependant, la décomposabilité garantit que si chacune des sous-formules d'une conjonction est cohérente, alors la conjonction des deux est nécessairement cohérente. On peut alors déterminer la cohérence de toutes ces sous-formules indépendamment. De ce fait, il existe un algorithme polynomial permettant de calculer la cohérence de toute représentation en DNNF. Ce langage satisfait donc **CO**.

Pour les transformations, c'est-à-dire les opérations effetuant une modification de la formule exprimée, c'est plus compliqué. En effet, certaines restrictions sur le langage de représentation vont faciliter certaines transformations quand d'autres vont les limiter. Il doit premièrement exister un algorithme polynomial en temps réalisant cette opération depuis ce langage de représentation ; et deuxièmement la forme d'arrivée doit appartenir au langage de départ. D'un côté l'ajout d'une restriction ne peut que permettre une transformation qui n'était alors pas réalisable, d'un autre côté, il n'est pas sûr qu'un algorithme réalisant une transformation respecte la nouvelle restriction imposée par le langage, et donc que la représentation retournée appartienne bien au langage de représentation désiré.

Prenons par exemple l'oubli (*forgetting*, ou **FO**). L'oubli d'un ensemble de variables X est une transformation qui renvoie une représentation α' de la formule α dans laquelle l'ensemble X de variables n'apparaît plus et tel que α' soit équivalent à $\exists X.\alpha$.

Pour les mêmes raisons que **CO**, cette transformation n'est pas réalisable en un temps polynomial sur une NNF, mais l'est sur une DNNF (chaque sous-formule d'une conjonction pouvant être traitée séparément). Cependant, si la formule était également déterministe (d-DNNF), elle peut ne plus l'être après traitement, ce qui fait que l'algorithme utilisé sur une DNNF ne fonctionne plus sur une d-DNNF (plus exactement, il fonctionne mais retourne une DNNF qui n'est pas forcément déterministe). NNF ne satisfait donc pas **FO**, DNNF le satisfait ,et d-DNNF ne le satisfait pas.

À la figure 2.6 est donnée la liste des requêtes et transformations étudiées dans Darwiche et Marquis [2002]. Les figures 2.7 et 2.8 recensent la satisfiabilité de chacune de ces requêtes et transformations dans les différents langages de représentation.

Les langages présentés ci-dessus sont considérés comme des classiques, des incontournables du genre, des blockbusters en quelque sorte. Cependant, ce ne sont pas les seuls à valoir le détour, en voici d'autres tout aussi intéressants.

Figure 2.6 – *Liste des requêtes et transformations. Toutes ces notions sont définies formellement en annexe.*

Notation	Query
CO	consistency
VA	validity
CE	clausal entailment
IM	implicant
EQ	equivalence
SE	sentential entailment
CT	model counting
ME	model enumeration

Notation	Transformation
CD	conditioning
FO	forgetting
SFO	singleton forgetting
∧C	conjunction
∧BC	bounded conjunction
∨C	disjunction
∨BC	bounded disjunction
¬C	negation

Figure 2.7 – *Requête pour les langages vus précédemment. √ signifie « réalisable en temps polynomial », et ○ signifie « non réalisable en temps polynomial sauf si $P = NP$ »*

L	CO	VA	CE	IM	EQ	SE	CT	ME
NNF	○	○	○	○	○	○	○	○
DNNF	√	○	√	○	○	○	○	√
d-NNF	○	○	○	○	○	○	○	○
d-DNNF	√	√	√	√	?	○	√	√
DNF	√	○	√	○	○	○	○	√
CNF	○	√	○	√	○	○	○	○
BDD	○	○	○	○	○	○	○	○
FBDD	√	√	√	√	?	○	√	√
OBDD	√	√	√	√	√	○	√	√
$\text{OBDD}_<$	√	√	√	√	√	√	√	√

Figure 2.8 – *Transformations pour les langages vu précédemment. √ signifie « réalisable en temps polynomial », • signifie « ne peut pas toujours être réalisé en espace polynomial » et ○ signifie « non réalisable en temps polynomial sauf si $P = NP$ »*

L	CD	FO	SFO	∧C	∧BC	∨C	∨BC	¬C
NNF	√	○	√	√	√	√	√	√
DNNF	√	√	√	○	○	√	√	○
d-NNF	√	○	√	√	√	√	√	√
d-DNNF	√	○	○	○	○	○	○	?
DNF	√	√	√	•	√	√	√	•
CNF	√	○	√	√	√	•	√	•
BDD	√	○	√	√	√	√	√	√
FBDD	√	•	○	•	○	•	○	√
OBDD	√	•	√	•	○	•	○	√
$\text{OBDD}_<$	√	•	√	•	√	•	√	√

OMDD

Un OMDD (diagramme de décision multivalué ordonné) est un OBDD (*read-once*, déterministe, ordonnée) dont le domaine des variables est discret mais non nécessairement booléen. Tout OMDD peut être transformé en OBDD en décomposant chaque variable non booléenne en un ensemble de variables booléennes. La transformation OMDD vers OBDD est donc linéaire [Fargier *et al.*, 2013a].

Tree-of-BDD, Tree-of-C

Un Tree-of-BDD [Subbarayan *et al.*, 2007] consiste en un arbre de décomposition arborescente, dont chaque sous-ensemble de variables (sommet de l'arbre de décomposition) est associé à un OBDD représentant la projection du problème initial sur les variables de ce sous-ensemble.

Un Tree-of-C [Fargier et Marquis, 2009] est la généralisation des Tree-of-BDD dans laquelle chaque sous-ensemble de variables est associé à une représentation dans un langage propositionnel quelconque (appelé langage C), et non obligatoirement par un OBDD. Ce langage C doit toutefois être un langage propositionnel complet, c'est-à-dire que toute formule de logique propositionnelle doit avoir une représentation dans ce langage C. Le langage C sera le plus souvent un DNNF, un OBDD ou $\text{OBDD}_{\leq}$.

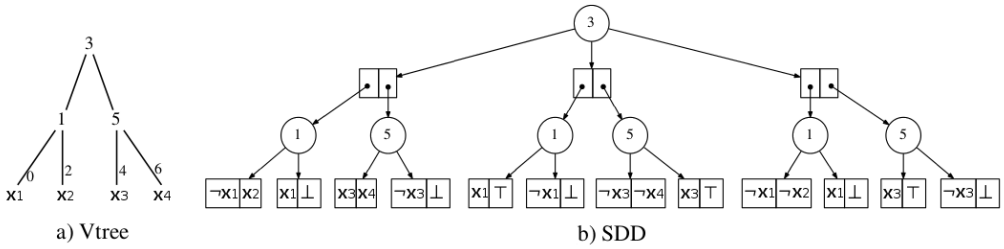
Dans [Fargier et Marquis, 2009], les auteurs montrent que malgré de bons résultats en termes de compacité, ce langage ne satisfait aucune des transformations étudiées dans le cadre de la carte de compilation, y compris le conditionnement (voir figure 2.10).

Figure 2.9 – *Requêtes et transformations réalisables en temps polynômial pour les langages Tree-of-OBDD et Tree-of-*$\text{OBDD}_{\leq}$*.* $\surd$ *signifie « satisfait »,* $\bullet$ *signifie « ne satisfait pas » et* $\circ$ *signifie « ne satisfait pas sauf si* $P = NP$ *».*

	CE	VA	CO	IM	EQ	SE	CT	ME
ToOBDD	$\circ$	$\surd$	$\surd$	$\surd$	?	$\circ$	?	$\surd$
$\text{ToOBDD}_{<}$	$\circ$	$\surd$	$\surd$	$\surd$	?	$\circ$	?	$\surd$
OBDD	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
$\text{OBDD}_{<}$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

	CD	FO	SFO	$\wedge$BC	$\wedge$C	$\vee$BC	$\vee$C	$\neg$C
ToOBDD	$\circ$	$\circ$	?	$\circ$	$\bullet$	$\circ$	$\circ$	$\circ$
$\text{ToOBDD}_{<}$	$\circ$	$\circ$	?	$\circ$	$\bullet$	?	$\circ$	$\circ$
OBDD	$\surd$	$\bullet$	$\surd$	$\circ$	$\bullet$	$\circ$	$\bullet$	$\surd$
$\text{OBDD}_{<}$	$\surd$	$\bullet$	$\surd$	$\surd$	$\bullet$	$\surd$	$\bullet$	$\surd$

Figure 2.10 – *SDD représentant la fonction* $(\neg x1 \wedge x2 \wedge x3 \wedge x4) \vee (x1 \wedge \neg x3 \wedge \neg x4) \vee (\neg x1 \wedge \neg x2 \wedge x3)$

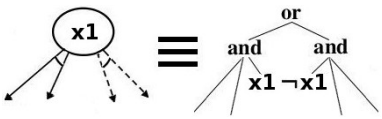


AOMDD (And/Or MDD) et *Tree-Driven* Automate

Les AOMDD [Mateescu et Dechter, 2006], ou *Tree-Driven* Automates [Fargier et Vilarem, 2004], sont des MDD dans lesquels, pour tout nœud de décision étiqueté par une variable x, chaque affectation possible de x peut avoir plusieurs arcs sortants associés, à condition que chacun des sous-diagrammes formés par ces arcs ne partagent aucune variable (propriété de décomposabilité) [1]. L'ensemble des arcs d'un même nœud associés à une même affectation de x est uni par l'opérateur « $\wedge$ ».

Le langage AOMDD intègre donc les deux opérateurs « $\wedge$ » et « $\vee$ », les nœuds de décision classiques étant constitués de l'opérateur « $\vee$ ».

Un AOMDD peut être écrit sous sa forme NNF en remplaçant chaque nœud de décision par un ensemble de nœuds « type NNF » de la façon suivante :



Notez toutefois que l'arité de l'opérateur « $\vee$ » n'est pas limitée à deux (MDD), de même que l'arité des opérateurs « $\wedge$ » pouvant être de deux (nœud de décision classique) ou plus.

SDD

Les SDD (*Sentential Decision Diagram*) [Darwiche, 2011] se situent quelque part entre les d-DNNF et les AOMDD. Contrairement aux AOMDD, l'opérateur « $\vee$ » n'est plus un nœud de décision, mais bien un « $\vee$ » sémantique. Un ordre partiel, sous la forme d'un *vtree** (arbre d'ordonnancement des variables), gouverne la décomposabilité.

1. Il faut rendre à ces arcs ce qui appartient à ces arcs.

2.1.3 Compilateurs

La compilation en langage booléen a déjà été étudiée. Nous avons donc à disposition un ensemble d'algorithmes et de compilateurs pour de nombreux langages. En voici un rapide résumé.

DNNF et d-DNNF

Dans [Darwiche, 1999], l'auteur vante les mérites du langages DNNF qui, comme le montre la carte de compilation, allie une certaine compacité et la satisfiabilité de la requête **CO** notamment. Il propose dans cet article des méthodes permettant une compilation exacte de certain problèmes, ainsi qu'une méthode d'approximation qui permet, lorsque la taille du problème est trop importante pour être compilée sans approximation, d'encadrer le problème Σ par deux représentation α et β tel que $\alpha \models \Sigma$ et $\Sigma \models \beta$.

Il propose également un compilateur *c2d*, mis à disposition en ligne, de CNF vers d-DNNF [Darwiche, 2004].

De même, Huang et Darwiche [2005a] proposent un compilateur descendant, DPLL with a Trace, permettant la compilation de problèmes SAT vers les langages d-DNNF, FBDD et OBDD.

BDD

Bryant [1986] utilise la propriété de conjonction bornée réalisable en temps polynomiale sur les $\text{OBDD}_{\leq}$ pour élaborer un brillant algorithme nommé *apply*. Cette algorithme permet la conjonction de deux $\text{OBDD}_{\leq}$, et permet donc une compilation ascendante, c'est-à-dire une compilation dans laquelle chaque élément (contraintes, sous-formule, ...) est écrit sous la forme d'un $\text{OBDD}_{\leq}$, opération généralement facile due au format et à la faible taille de ces éléments, et on réalise ensuite la conjonction de ces éléments.

De nombreux utilitaires permettent la manipulation d'OBDD. Nous pouvons citer par exemple, et par ordre chronologique, ABCD [Biere, 2000], BuDDy [Lind-Nielsen, 2002], JDD [Vahidi, 2003], la très classique et très complète CUDD [Somenzi, 2005] qui permet également la manipulation d'ADD (voir §2.2.1), JavaBDD [Whaley, 2007] et Crocopat [Beyer, 2008].

Différentes approches de compilation impliquant des diagrammes de décision ont également été implémentées. Nous pourrons noter par exemple Vempaty [Vempaty, 1992] qui compile des réseaux de contraintes en OMDD en utilisant une procédure ascendante. Approche reprise et implémentée au cours de sa thèse par Amilhastre [Amilhastre, 1999].

2.2 Compilation valuée

Cohérence et conditionnement sont des requêtes qu'un langage doit impérativement satisfaire pour être considéré comme utilisable pour la configuration de produits. Cependant, on peut apporter d'autres informations fort utiles à l'utilisateur que la simple faisabilité du produit en cours de configuration. En effet, l'utilisateur est généralement intéressé par un retour sur le prix qu'il peut espérer du produit en cours de configuration, ou du moins le prix d'un produit de coût minimal correspondant à sa configuration partielle courante. On peut aussi imaginer aussi une information sur un délai de production, ou encore une utilité quelconque. Même si on peut généralement calculer facilement, dans quasiment tous les langages de représentation, ces informations sur un produit fini, guider l'utilisateur sur des choix lui permettant d'atteindre une configuration optimisant la valuation finale est un problème NP-difficile si le langage de représentation n'intègre pas cette information de valuation.

Le but est donc d'étudier des langages de représentation de fonctions à valeur dans une structure de valuation plus riche que $\{\top, \bot\}$, qui sont des généralisation des langages booléens vu précédemment. Ainsi le domain d'arrivée de ces fonctions ne sera pas limité à $\mathbb{B}$, mais généralement à $\mathbb{N}$ ou $\mathbb{R}$. Nous conservons les propriétés qui font l'intérêt de ces langages en passant dans le domaine valué, et nous étudions l'usage qui peut être fait de l'ajout de ce système de valuation plus riche.

2.2.1 Langages

Les langages que nous considérons sont des généralisations à un domaine non booléen des langages précédents. Nous ne nous cantonnons plus à un système booléen limité aux valeurs « $\top$ » et « $\bot$ » et à leurs opérateurs associés « $\wedge$ » et « $\vee$ », mais à une structure de valuation $\mathcal{E} = \langle E, \geq, OP \rangle$, avec E ordonné selon un ordre $\geq$ (en général nous prendrons pour E les ensembles $\mathbb{R}^+, \mathbb{N}^+$, [0,1], ou encore $\mathbb{Q}^+$ afin de travailler avec des valeurs exactes), et OP un ensemble d'opérateurs (en général $\times$ et $+$, éventuellement min et max).

De plus, dans les langages qui vont suivre, les variables et opérateurs utilisés ne sont pas nécessairement binaires, cela étant particulièrement pratique pour représenter des variables ayant un domaine à valeurs discrètes non nécessairement booléennes.

Le passage des NNF aux VNNF (*Valued NNF*) nous permet donc de passer de représentations d'applications ayant pour profil $\mathbb{B}^* \to \mathbb{B}$ à des représentations d'applications ayant pour profil $\mathbb{N}^* \to \mathbb{R}$.

VNNF*

Une représentation en VNNF [Fargier et Marquis, 2007] est un DAG (graphe dirigé acyclique) défini par une structure de valuation $\mathcal{E} = \langle E, \geq, OP \rangle$, où chaque nœud non terminal est étiqueté par opérateur de OP, chaque nœud pouvant avoir un nombre quelconque de fils, et les nœuds terminaux sont étiquetés soit par une valeur de E soit par une fonction locale*.

Une NNF est alors une VNNF particulière dans laquelle la structure de valuation est booléenne, c'est-à-dire que E est égal à « $\top$ » ou « $\bot$ », l'opérateur t-norm est « $\wedge$ », t-conorm est « $\vee$ », et les fonction locales sont les littéraux.

On retrouve avec ce langage les mêmes propriétés générales que dans celui des NNF : c'est un langage compact, mais ne satisfaisant que peu de requêtes.

VCSP*

Un VCSP (pour Valued CSP) [Schiex *et al.*, 1995; Bistarelli *et al.*, 1999] est un CSP auquel on ajoute pour chaque contrainte une valuation représentant le « coût » de violation, c'est-à-dire le coût qu'il faudra acquitter si cette contrainte est violée. La résolution du CSP ne consiste plus en trouver une solution satisfaisant chacune des contraintes, mais les solutions minimisant une combinaison des valuations associées aux contraintes violées.

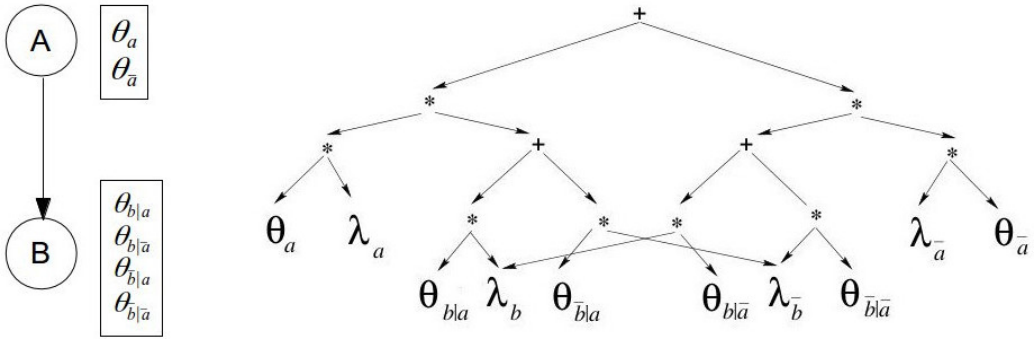
WCNF*

Sur le même concept que les VCSP, une valuation, correspondant au coût qu'il faudra acquitter si cette clause n'est pas respectée, est associée à chacune des clauses [Pinkas, 1991].

AC*

Les circuits arithmétiques [Darwiche, 2003] sont l'équivalent dans un domaine valué des d-DNNF. Les nœuds non terminaux sont étiquetés par les opérateurs « $+$ » et « $\times$ », et les nœuds terminaux sont étiquetés soit par une valeur réelle, soit par une fonction locale. Ils reprennent le principe de déterminisme et de décomposabilité mais appliqué au domaine valué (pour chaque opérateur « $+$ », au maximum une seul sous-formule peut être différente de 0, et pour chaque opérateur « $\times$ » les sous-formules ne partagent aucune variable) (exemple figure 2.11).

Figure 2.11 – *Exemple d'un circuit arithmétique (à droite) représentant un réseau bayésien (à gauche). les θ correspondant à des probabilités, et les λ étant des fonctions locales à valeur dans* $\{0, 1\}$



On retrouve également toutes les caractéristiques du langage d-DNNF, ce qui en fait un langage à la fois succinct et complet.

VDD

Nous voici enfin arrivés aux diagrammes de décision. Comme ces langages vont être largement abordés dans le chapitre suivant, nous ne verrons ici que le principe des VDD.

Les VDD constituent une généralisation à un domaine valué des OBDD. Ils sont donc ordonnés et les variables sont à domaine discret. Ils sont définis par une structure de valuation $\mathcal{E} = \langle E, \otimes, \succ \rangle$, où des valeurs de E peuvent être placées sur les nœuds comme sur les arcs, et les nœuds terminaux « $\top$ » et « $\bot$ » sont remplacés par un ou plusieurs nœuds terminaux portant une valeur de E. Un opérateur $\otimes$ est défini permettant d'agréger les éventuelles valuations étiquetant les arcs et les nœuds afin d'obtenir la valuation finale d'un chemin.

On trouve plusieurs sous-langages dans la famille VDD :

- le langage ADD (*Algebraic Decision Diagram*) introduit par Bahar *et al.* [1993] est le plus proche du langage OBDD. Il n'y a pas de valuation sur les arcs, et seuls les nœuds terminaux sont étiquetés par une valuation.
- le langage SLDD (*Semi ring Labelled Decision Diagram*), introduit par Wilson [2005], et qui ne comporte qu'un seul nœud terminal « neutre », et dont les arcs sont étiquetés par des valuations, valuations agrégées par un opérateur à définir.
- le langage AADD (*Affine Algebraic Decision Diagram*), introduit par Tafertshofer et Pedram [1997] sous le nom de FEVBDD (*Factored Edge*

Valued Binary Decision Diagram) puis revu par Sanner et McAllester [2005], qui ne comporte qu'un seul nœud terminal, et dont les arcs sont étiquetés par un couple de valuations $\langle f, q \rangle$, l'agrégation alliant les opérateurs $+$ et $\times$.

Nous pouvons également citer d'autres langages valués basés sur les diagrammes de décision, mais ceux-ci ne seront pas étudiés ici car le langage SLDD permet plus de liberté, tout en gardant les mêmes propriétés, que ces langages. Nous avons :

- le langage EVBDD* (*Edge Valued Binary Decision Diagram*), introduit par Lai et Sastry [1992], qui peut être vu comme un OBDD auquel on peut ajouter des poids sur les arcs.
- le langage MDD valué qui consiste en une utilisation du MDD en ajoutant des poids aux arcs.
- Le langage des AOMDD pondérés [Mateescu et Dechter, 2007] qui sont une extension des AOMDD. Une valuation est associée à chaque arc, l'opérateur « $\wedge$ » des AOMDD non valués est remplacé par un opérateur $\otimes$, et ce même opérateur est utilisé comme opérateur d'agrégation des valuations.
- le langage PSDD (Probabilistic Sentential Decision Diagrams) [Kisa *et al.*, 2014] est une extention du langage SDD auquel on ajoute des valuations, correspondant à des probabilités, sur les arcs.

2.2.2 Compilateurs

Plusieurs compilateurs ont été développés dans le cadre valué.

AC

Sur les mêmes bases que les d-DNNF, Darwiche [2003] donne des méthodes de compilation de réseaux bayésiens en circuits arithmétiques. Ces méthodes sont principalement dédiées aux réseaux bayésiens, pour lequel elles sont particulièrement utiles.

VDD

Sanner et McAllester [2005] établissent une version valuée de l'algorithme *apply* de Bryant [1986] également appelée *apply*[2] qui réalise l'addition, la multiplication, la différence ou la division (équivalent à la conjonction) de deux

2. Les compétences permettant la compréhension de cet algorithme tiennent toutefois plus de l'égyptologie ancienne que de l'algorithmique.

AADD, et permettant une compilation ascendante. Cette tâche est rendue plus ardue par le fait que, comme nous le verrons plus tard, les AADD ne satisfont pas la transformation de l'addition bornée.

Sanner et McAllester ont également implémenté un compilateur qu'ils ont testé sur des réseaux bayésiens. Cependant, nous n'avons pas réussi à accéder à ce compilateur, ce qui nous a empêchés de le tester.

Hadzic [2004] propose une méthode de compilation ascendante en une structure qu'il appelle *AugmentedLabelledGraph*, très proche des EVBDD. La méthode développée permet également de limiter la recherche de solution à un maximum ou un minimum de prix.

Hadzic et O'Sullivan [2009] proposent une méthode ascendante de compilation en MDD valué.

Un compilateur en AOMDD a également été implémenté par Marinescu [Mateescu *et al.*, 2008]. Le programme permet la compilation de WCSP et de réseaux bayésiens avec une approche ascendante. Toutefois, ce programme ne permet pas d'intégrer, au WCSP à compiler, des contraintes « dures » (non valuées).

Applications

Hoey *et al.* [1999] propose l'algorithme SPUDD (*Stochastic Planning Using Decision Diagram*), qui compile des MDP (*Markov decision process*), utilisé en planification, en deux ADD représentant probabilité d'une action et récompense. Cet algorithme est utilisé en planification, mais aussi pour la résolution de MDP en général.

En diagnostic, domaine de l'intelligence artificielle consistant à statuer sur le bon fonctionnement d'un système, ou sinon, déterminer les causes et origines du mauvais fonctionnement, la compilation vers des langages booléens est également utilisée. Dans [Sztipanovits et Misra, 1996] et [Torasso et Torta, 2003] par exemple, les auteurs utilisent des OBDD comme langage cible, quand dans [Darwiche, 2001a] et [Huang et Darwiche, 2005b], les DNNF sont mis à l'honneur.

Toujours avec un domaine d'arrivée booléen, Niveau *et al.* [2010] utilisent la compilation pour des problèmes de planification. Ici, le problème est compilé en automate à intervalles, un langage proche des MDD permettant la compilation de problèmes dont les variables ne sont pas nécessairement discrètes.

Enfin, et toujours dans un domaine d'arrivée booléen, la compilation est également déjà utilisée pour résoudre des problèmes de configuration de produits. Les auteurs de [Sinz, 2002] et [Hadzic *et al.*, 2007] utilisent respectivement comme langage de compilation le langage PI (prime Implicates, sous-langage de la famille des CNF), et les MDD. Amilhastre *et al.* [2002] et Pargamin [2003] utilisent eux respectivement des automates à état fini et des « cluster-trees » afin de représenter leurs données compilées.

Chapitre 3

Diagrammes de décision valués

Dans cette partie, nous approfondissons l'étude de trois langages de représentation, tous sous-ensembles de ce que nous appellons les Diagrammes de Décision Valués (VDD). Ces VDD sont ordonnés selon « $\leq$ ». On peut donc voir les VDD comme l'extention au domaine valué des $\text{OBDD}_{\leq}$.

Dans le domaine booléen, la carte de compilation nous indique que les requêtes principales sont toutes réalisables en un temps polynomial dans la taille de la représentation, la plupart étant même réalisables en un temps linéaire dans la taille de la représentation, linéaires dans le nombre de variables voire en temps constant (la requête cohérence par exemple). Les transformations nous assurent entre autres la capacité de construire (i.e. compiler) notre problème, et de le manipuler facilement une fois compilé.

Ce chapitre est divisé en trois parties. La première est consacrée à la définition formelle de ces langages. Nous y verrons d'abord des notions générales, des notations concernant notamment l'écriture de diagrammes de décision, mais aussi d'un système de valuation, ainsi que les différents types de VDD dont cette thèse fait l'objet, tels qu'ils ont été introduits, puis tels que nous les voyons. Nous verrons ensuite certaines propriétés de ces langages telles que leur normalisation, leur compacité théorique ou les traductions d'un langage à l'autre. Finalement, nous aborderons l'aspect pratique en détaillant quelque peu l'implémentation de notre compilateur de ces langages.

3.1 Définitions générales et notations

3.1.1 Notations

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_i} ; nous noterons D_X l'ensemble des affectations $\vec{x}$ des variables de X.

Système de valuation

Une structure de valuation $\mathcal{E}$ est un triplet $\langle E, \otimes, \succ \rangle$ où E est un ensemble de valuations partiellement ou totalement ordonné selon $\succ$, une relation transitive et irréflexive sur E, avec un élément maximum selon $\succ$ (élément optimum) noté $\top$ et un élément minimum selon $\succ$ noté $\bot$. $\otimes$ est un opérateur associatif et binaire sur E et vérifiant les propriétés de neutralité de $\top$, de monotonie, et dont $\bot$ est un élément absorbant.

Langage de représentation (inspiré de [Gogic *et al.*, 1995])

Soit une structure de valuation $\mathcal{E}$, un *langage de re présentation* $\mathcal{L}$ portant sur X valué dans $\mathcal{E}$ est un 4-uplet $\langle C_\mathcal{L}, Var_\mathcal{L}, f^\mathcal{L}, s_\mathcal{L} \rangle$ où

- $C_\mathcal{L}$ représente un ensemble de structures de données α (aussi appelées formule $C_\mathcal{L}$)
- $Var_\mathcal{L}$ est une fonction associant à chaque α de $C_\mathcal{L}$ l'ensemble X de variables dont il dépend
- $f^\mathcal{L}$ (souvent simplifié par f lorsque le langage $\mathcal{L}$ utilisé ne fait aucune ambiguïté) est une fonction d'interprétation permettant, à toute structure de données α de $C_\mathcal{L}$, d'associer à toute affectation de $Var_\mathcal{L}(\alpha)$ une valuation de E
- $s_\mathcal{L}$ est une fonction donnant la taille de la structure de donnée α dans $\mathbb{N}$.

Nous pouvons détacher ici la notion de *sémantique* $f^\mathcal{L}$ d'un langage, qui correspond à l'interprétation qui peut (doit) être faite d'une structure de données quelconque exprimée dans ce langage, et la notion de *représentation* $C_\mathcal{L}$ d'une fonction faite d'un problème dans ce langage, autrement dit la syntaxe de ce langage.

Formules équivalentes

Deux formules $\alpha \in \mathcal{L}_1$ et $\beta \in \mathcal{L}_2$, avec éventuellement $\mathcal{L}_1 = \mathcal{L}_2$ mais pas nécessairement, sont équivalentes si $Var_{\mathcal{L}_1}(\alpha) = Var_{\mathcal{L}_2}(\beta)$ et si $f^{\mathcal{L}_1}(\alpha) = f^{\mathcal{L}_2}(\beta)$. C'est-à-dire si α et β portent sur les mêmes variables, et si tout affectation $\vec{x}$ de $Var_{\mathcal{L}_1}(\alpha)$ (ou $Var_{\mathcal{L}_2}(\beta)$) conduira à $f^{\mathcal{L}_1}(\alpha)(\vec{x}) = f^{\mathcal{L}_2}(\beta)(\vec{x})$

Diagramme de décision valué

Un *diagramme de décision valué* (VDD) α est une structure de données permettant de représenter une fonction f_α 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.

Un VDD est un DAG 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 $\mathcal{E}$.

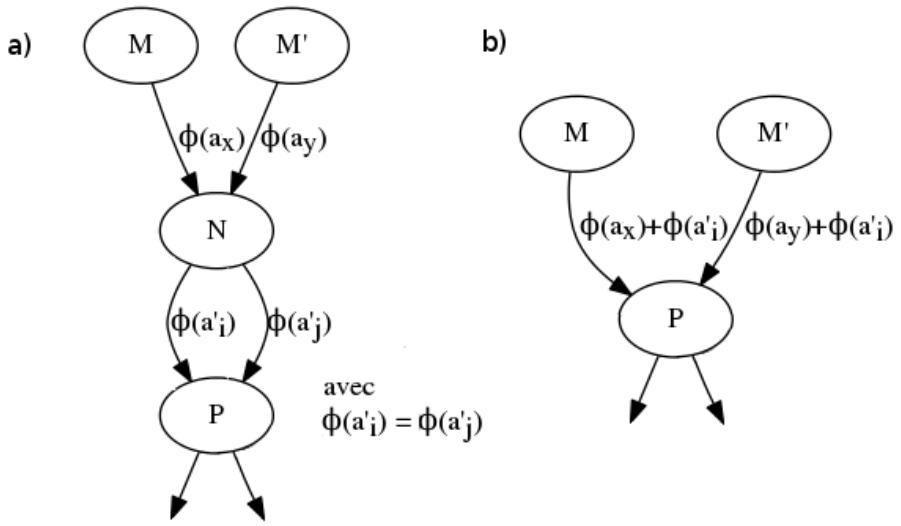
On note $In(N)$ (resp. $Out(N)$) l'ensemble des arcs entrant dans (resp. issus de) N et $In(a)$ (resp. $Out(a)$) le nœud origine (resp. l'extrèmité) de a. On note également $Out_i(N)$ l'arc issu de N tel que $val(Out_i(N)) = d_i$. De même, dire qu'un arc a pointe sur un nœud N signifie que $Out(a) = N$.

Un VDD est *ordonné* : 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.

Nous considérons dans la suite plusieurs langages de représentation, sous-langages de la famille VDD, à savoir ADD, SLDD et AADD. Chacun d'entre eux impose des restrictions sur les diagrammes admissibles (par exemple, dans le langage ADD, seuls les nœuds terminaux portent des valuations ϕ), et définit comment ils sont interprétés. De ce fait, les langages ADD, SLDD et AADD diffèrent à la fois syntaxiquement (par la façon dont les arcs et les nœuds sont étiquetés) et sémantiquement (par la façon dont les formules sont interprétées).

Dans tous les cas la fonction $Var_{\mathcal{L}}(\alpha)$ d'un VDD α quelconque retourne l'ensemble des variables $Var(\alpha)$ et la fonction taille $s_{\mathcal{L}}(\alpha)$ retourne la taille du diagramme de décision, c'est-à-dire le nombre de nœuds et/ou le nombre d'arcs. $C_{\mathcal{L}}$ et $f^{\mathcal{L}}$ sont à définir pour chaque langage sous-ensemble de la famille VDD.

Figure 3.1 – *exemple d'un SLDD$_+$ avec : en a) un nœud N bégayant, en b) le nœud N est supprimé.*



Nœud bégayant

Un nœud bégayant est un nœud n'apportant rien au schmilblick, c'est-à-dire un nœud dont tous les arcs sortants se valent car ils portent la même valuation ϕ (si ils en ont une) et pointent sur le même nœud. Autrement dit, soit N un nœud, a_i et a_j deux arcs tels que $a_i = Out_i(N)$ et $a_j = Out_j(N)$, le nœud N est dit bégayant si, $\forall i, j$, $\phi(a_i) = \phi(a_j)$ et $Out(a_i) = Out(a_j)$. Un exemple de nœud bégayant est donné figure 3.1.

Ces nœuds bégayants sont gardés ou supprimés suivant les cas. La suppression se fait en reconnectant le ou les arcs entrant dans notre nœud bégayant, directement sur le nœud suivant, en agrégant la valuation des arcs sortants aux arcs entrants. C'est-à-dire, pour tout arc $a = In(N)$, et avec $a' = Out(N)$ (tous les arcs $Out(N)$ issue du nœud bégayants N se valent), alors on obtient $Out(a) \leftarrow Out(a')$ et $\phi(a) \leftarrow \phi(a) \otimes \phi(a')$, $\otimes$ dépendant du langage [1] (Exemple figure 3.1).

Dans souci d'uniformisation, le choix de garder ou non l'ensemble des nœuds bégayants doit être identique sur l'ensemble du diagramme. Généralement, ces nœuds sont conservés en cours de construction ainsi que pour certaines transformations et traitements, et sont supprimés lors de la construction de la représentation finale.

1. Notez que si le diagramme est normalisé, alors $\phi(a')$ sera égale à l'élément neutre.

Solution non admissible et élément absorbant

Le passage du monde booléen au monde valué ne nous libère pas de la notion de solution non admissible (correspondant au nœud « faux », ou « $\perp$ », d'un BDD). Dans les langages ADD et SLDD, cette notion est généralement liée au système de valuation. Par exemple lorsque l'on veut exprimer une fonction de coût, une solution non admissible correspondra à un coût infini. Il en va de même pour tout système de valuation de nature additive, et que l'on cherche à minimiser. À l'inverse, pour un système de valuation de nature multiplicative (comme par exemple une probabilité), une solution non admissible correspond à une valuation de 0.

Les AADD ne possèdent pas dans leurs valuation d'élément absorbant. Nous rajoutons donc pour nos besoins l'élément de valuation « $\perp$ » que nous définirons en même temps que le langage AADD.

Dans un souci d'uniformisation, tout arc valué par l'élément absorbant doit être directement relié au nœud terminal.

Nœuds isomorphes

Deux nœuds N et N' d'un même diagramme de décision sont dit isomorphes s'ils portent tous deux sur la même variable, portent la même valuation (si ils en ont une), et si chacun des arcs sortants a_i et a'_i porte la même valuation et point sur un même nœud. Autrement dit si $Var(N) = Var(N')$, $\phi(N) = \phi(N')$, et si $\forall i$ nous avons $\phi(a_i) = \phi(a'_i)$ et $Out(a_i) = Out(a'_i)$, avec $a_i = Out_i(N)$ et $a'_i = Out_i(N')$.

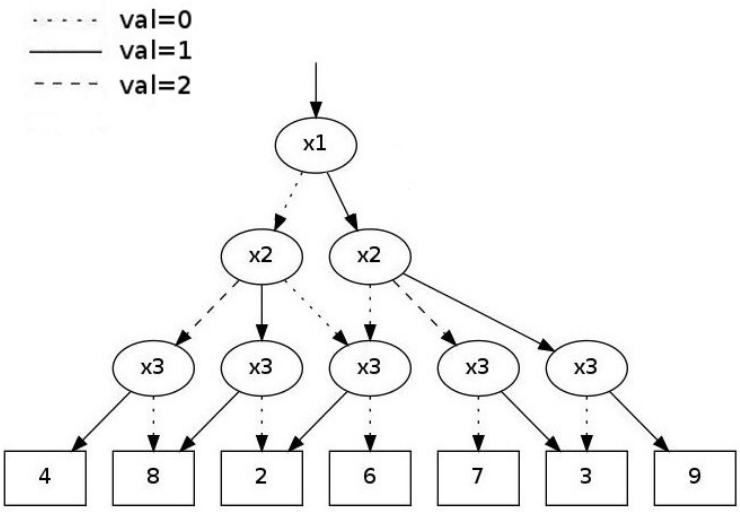
Des nœuds isomorphes sont donc identiques et représentent la même fonction, ils peuvent être fusionnés en un seul. Le nœud résultant de cette fusion reçoit alors l'ensemble des arcs entrants.

Pour un exemple de fusion de nœuds isomorphes se reporter à l'exemple de normalisation, figure 3.9, étape 4.

Forme réduite

Un diagramme de décision valué est dit sous forme réduite s'il ne contient pas de nœuds isomorphes ni de nœuds bégayants. Tout VDD ordonné possède une unique forme réduite, qu'il est possible d'obtenir en temps linéaire en sa taille. Cependant, pour certain langages, le processus de réduction peut nécessiter une phase de normalisation, que nous approfondirons plus tard. Nous supposons par la suite que les diagrammes de décision considérés sont sous forme réduite.

Figure 3.2 – *Exemple de ADD. Chaque arc issu d'un nœud étiquetté par une variable x_i correspond à une valeur possible du domaine de cette variable.*



3.1.2 Diagrammes de décision algébriques (*Algebraic Decision Diagrams*)

Le langage ADD [Bahar *et al.*, 1993] est une 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.

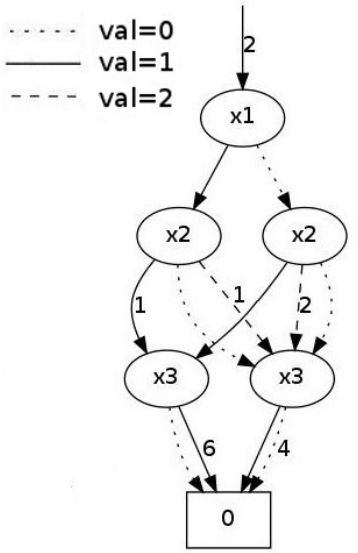
Définition

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

- si α est un nœud terminal N , alors $f_\alpha(\vec{x}) = \phi(N)$
- sinon, la racine N de α 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 β le ADD de racine M dans α ; 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 sera représentable en espace polynomial par un VCSP fortement additif, prend 2^n valeurs différentes, d'où une taille exponentielle des ADD qui la représentent (exemple figure 3.2).

Figure 3.3 – *Exemple d'un SLDD$_+$ représentant la même fonction que le ADD donné à la figure 3.2.*



3.1.3 Diagrammes de décision valués dans un semi-anneau (*Semiring Labeled Decision Diagrams*)

Dans le cadre SLDD [Wilson, 2005] ce sont les arcs, et non les nœuds terminaux, qui sont étiqueté par les valeurs de E. 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'agrégation des valuations d'un chemin est faite via l'opérateur $\otimes$ de notre semi-anneau.

Définition

Un SLDD α 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 $\mathcal{E} = \langle E, \otimes, \oplus, 1_s, 0_s \rangle$ est un semi-anneau. Un SLDD associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x})$ appartenant à E définie par :

- si α est le nœud terminal alors $f_\alpha(\vec{x}) = 1_s$
- sinon, la racine N de α 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 β le SLDD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = \phi(a) \otimes f_\beta(\vec{x})$.

On associer à α une valeur $\phi_0 \in E$ (son *offset*). La fonction « augmentée » f_{α,ϕ_0} que représente α ainsi décoré par ϕ_0, est définie par : pour tout $\vec{x} \in D_X$, $f_{\alpha,\phi_0}(\vec{x}) = \phi_0 \otimes f_\alpha(\vec{x})$.

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 ($\oplus$ n'est utilisé que pour l'utilisation et l'exploitation d'un SLDD, par exemple lorsque l'on veut calculer une valuation optimale, ou lorsque l'on veut éliminer une ou plusieurs variables). Pour cette raison, nous avons proposé et 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.*, 2013c]. On notera $\text{SLDD}_\otimes$ le langage SLDD utilisant $\otimes$ comme opérateur d'agrégation des valuations.

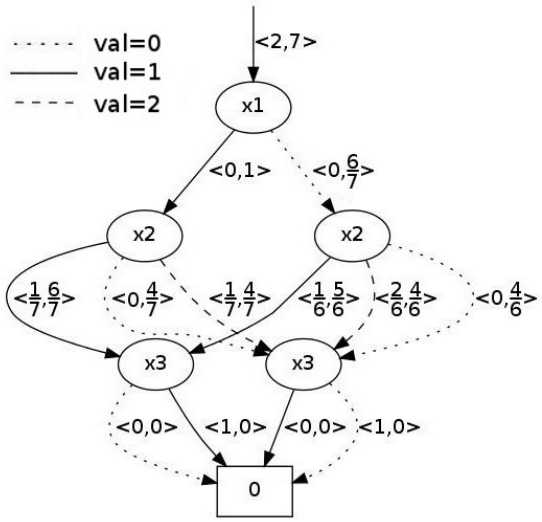
Deux monoïdes sont particulièrement intéressants : $\mathcal{E} = \langle \mathbb{R}^+ \cup \{+\infty\}, +, 0 \rangle$ pour tous les problèmes dont les valuations sont de nature additive (coûts, utilités, ...) et $\mathcal{E} = \langle \mathbb{R}^+, \times, 1 \rangle$ pour tous les problèmes dont les valuations sont de nature multiplicative (probabilités). Les langages associés sont notés respectivement SLDD_+ et $\text{SLDD}_\times$. Chacun admet un élément absorbant (0 pour $\text{SLDD}_\times$, $+\infty$ pour les SLDD_+),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 ».

Notons que dans sa définition, Wilson [2005] ne force pas le langage SLDD à être ordonné, mais seulement *read-once*. Dans chaque chemin d'un SLDD, une variable ne peut être rencontrée qu'une seul fois. Cependant, 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) (voir un exemple de SLDD_+ à la figure 3.3).

3.1.4 Diagrammes de décision algébriques affines (*Affine Algebraic Decision Diagrams*)

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

Figure 3.4 – *Exemple d'un AADD représentant la même fonction que les ADD et SLDD donnés aux figure 3.2 et 3.3.*



Définition

Un AADD α 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 $\mathbb{R}^+$. α associe à tout $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in \mathbb{R}^+$ définie par :

- si α est le nœud terminal N, $f_\alpha(\vec{x}) = 0$
- sinon, la racine N de α 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 $val(a) = d$, $\phi(a) = \langle b, c\rangle$ le couple de valeurs associée à a, et β le AADD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = b + (c \times f_\beta(\vec{x}))$.

On associe à α un couple $\langle b_0, c_0\rangle$ de $\mathbb{R}^+ \times \mathbb{R}^+$ (son *offset*). La fonction « augmentée » $f_{\alpha,\langle b_0,c_0\rangle}$ que représente α ainsi décoré par $\langle b_0, c_0\rangle$, est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\langle b_0,c_0\rangle}(\vec{x}) = b_0 + (c_0 \times f_\alpha(\vec{x}))$.

L'élément neutre du système de valuation utilisé par le AADD est $\langle 0, 1\rangle$ (voir un exemple de AADD à la figure 3.4).

Grosso modo, un SLDD_+ peut être considéré comme un AADD particulier dont les facteurs multiplicatifs f sont égaux à 1 (l'élément neutre de l'opérateur multiplication). De la même façon, un $\text{SLDD}_\times$ peut être grosso modo considéré comme un AADD particulier dont les facteurs additifs q sont nuls.

Pour nos besoins, nous ajoutons la valuation « $\perp$ » au système de valuation (maintenant défini sur $(\mathbb{R}\times\mathbb{R})\cup\{\perp\}$) tel que $\forall x$ on a $x+\perp = \perp+x = \perp$ et $x \times \perp = \perp \times x = \perp$. cet élément permet de représenter la notion d'impossibilité (coût infini, probabilité nulle, ...).

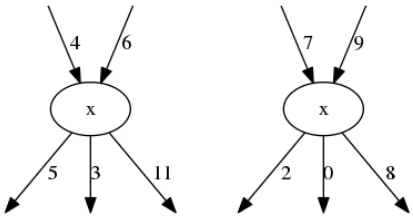
Le langage AADD est muni d'une procédure de normalisation qui permet de garantir, pour chaque fonction à représenter et étant donné un ordre de variables, une représentation unique et minimale.

3.1.5 Forme normalisée

Lors de la compilation, ou de la représentation de données, disposer d'une représentation canonique est une propriété utile. En plus d'être de taille minimale, cette représentation est aussi unique, ce qui permet de facilement détecter les sous-formules identiques.

Parmi les trois langages ci-dessus, la réduction seule ne permet de garantir la canonicité que dans le cas du langage ADD. En effet, dans le cas des langages SLDD et AADD, il existe plusieurs représentations d'une même fonction, comme le montre la figure 3.5 représentant deux nœud SLDD$_+$ strictement équivalents.

Figure 3.5 – *Deux nœuds SLDD$_+$, avec leurs arcs entrants et sortants, strictement équivalents.*



Pour pallier ce problème, la solution consiste à respecter une norme portant sur l'ensemble des valuations portées par l'ensemble des arcs sortants d'un même nœud.

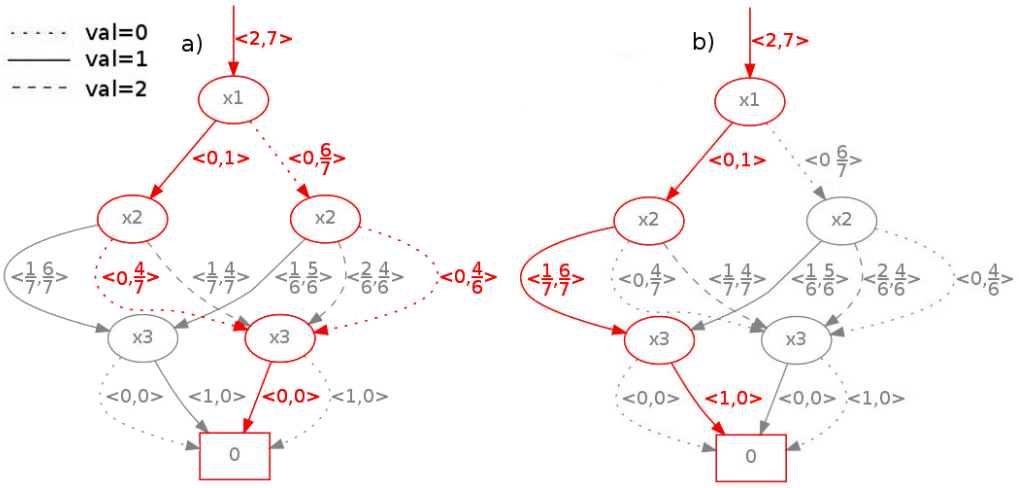
C'est ce qu'ont fait Sanner et McAllester [2005] avec les AADD. En effet, ne sont considérées comme AADD que les représentations correspondant à la définition donnée ci-dessus, et respectant les règles ci-dessous.

AADD sous forme normalisée

Un AADD est normalisé ssi pour tout nœud N d'arcs sortants $Out_i(N) = a_i$, avec $\phi(a_i) = \langle b_i, c_i \rangle$ et $\phi(a_i) \neq \bot$

- $min_i(b_i) = 0$
- $max_i(b_i + c_i) = 1$
- si $\forall \vec{x} \in D_X$, $f_{Out(a_i)}(\vec{x}) = 0$ alors $c_i = 0$

Figure 3.6 – *En a) les deux chemins de valuation minimale (pour chaque arc, $b = 0$), en b) le chemin de valuation maximale (pour chaque arc, $b + c = 1$). On a* $\min_{\vec{x}} f(\vec{x}) = 2$ *et* $\max_{\vec{x}} f(\vec{x}) = 9$.



Les deux premiers points permetent l'harmonisation des valuations des arcs sortant d'un nœud. Le troisième point est nécessaire car si un arc pointe sur une sous-fonction nécessairement égale à zéro, alors n'importe quelle valuation pourrait être affectée à c sans aucune influence sur la fonction d'interprétation. On perdrait alors l'unicité de la représentation, c'est pourquoi on le fixe à la valeur 0.

Cette forme normalisée, en plus de garantir l'unicité de la représentation et une taille minimale possède une propriété intéressante. En effet, elle nous donne la garantie que, quel que soit le nœud, un arc de valuation $\langle b, c\rangle$ avec $b = 0$ existe, et donc qu'il existe pour tout AADD α au moins un chemin de la racine au nœud terminal dont toutes les valuations b sont égales à 0, et que ce(s) chemin(s) peu(ven)t être trouvé(s) en un temps linéaire dans la taille de $Var(\alpha)$, soit sans aucun retour en arrière. Ce ou ces chemins correspondent bien entendus à la solution de coût minimal, et ce coût est égal à l'offset b_0. On a donc $min_{\vec{x}}(f_\alpha(\vec{x})) = b_0$.

La normalisation nous donne également la garantie qu'il existe toujours un arc étiqueté par $\langle b, c\rangle$ dont la somme $b + c$ vaut 1, ceci allié au fait que l'on peut toujours finir un chemin par un arc de valuation $\langle 1, 0\rangle$ nous assure de trouver, en temps linéaire dans la taille de $Var(\alpha)$, un (le) chemin de coût maximal, le coût étant égal à $b_0 + c_0$. On a donc $max_{\vec{x}}(f_\alpha(\vec{x})) = b_0 + c_0$ (voir figure 3.6).

On peut appliquer le même raisonnement aux SLDD. Nous proposons donc une définition similaire (quoique plus simple) d'une forme normalisée pour les SLDD_+ et $\text{SLDD}_\times$.

SLDD sous forme normalisée
Un SLDD_+ est normalisé ssi pour tout nœud N d'arcs sortants $Out_i(N) = a_i$, on a $min(\phi(a_i)) = 0$.
Un $\text{SLDD}_\times$ est normalisé ssi pour tout nœud N d'arcs sortants $Out_i(N) = a_i$, on a $max(\phi(a_i)) = 1$.

Nous retrouvons des propriétés similaires aux AADD. Pour un SLDD_+, nous avons $min_{\vec{x}}(f_\alpha(\vec{x})) = \phi_0$, avec ϕ_0 son offset. Cette ou ces solutions peuvent être obtenue(s) en suivant systématiquement le chemin passant par un arc de valuation $\phi = 0$.
Pour un $\text{SLDD}_\times$, nous avons $max_{\vec{x}}(f_\alpha(\vec{x})) = \phi_0$, avec ϕ_0 son offset. Cette ou ces solutions peuvent être obtenue(s) en suivant systématiquement le chemin passant par un arc de valuation $\phi = 1$.

3.1.6 e-SLDD

Dans [Fargier *et al.*, 2013c], nous avons proposé un nouveau cadre correspondant à une extension du cadre SLDD défini ci-dessus, le *e*-SLDD (pour *extended SLDD*). Dans le cadre *e*-SLDD, nous ajoutons au cadre déjà existant quelques éléments manquants que nous avons pointés précédemment.

Tout d'abord, nous avons étendu le système de valuation $\mathcal{E}$ à un monoïde $\langle E, \otimes, 1_s\rangle$. Comme expliqué précédemment, l'opérateur $\oplus$ de la structure de valuation originale $\mathcal{E} = \langle E, \otimes, \oplus, 1_s, 0_s\rangle$ n'apporte rien d'un point de vue représentation. Nous avons également ajouté, à l'instar du AADD, une procédure de normalisation, qui permet une représentation unique et minimale.

$\oplus$-normalisation et $\oplus$-réduction
Une formule α est $\oplus$-normalisée ssi pour tout nœud N de α et avec $a_i = Out_i(N)$, $\oplus(\phi(a_i)) = 1_s$. Une formule α est $\oplus$-réduite ssi elle est $\oplus$-normalisée et réduite.

$\oplus$ doit bien sûr être commutatif et idempotent : commutatif car l'ordre des arcs ne doit pas importer, idempotent car appliquer cette opération sur un nœud déjà normalisé ne doit pas en changer le résultat. L'opérateur $\otimes$ doit également être distributif à gauche par rapport à $\oplus$. C'est-à-dire avec a, b et $c \in E$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$.

Dans le cas du SLDD$_+$ (resp. SLDD$_\times$), l'opérateur $\oplus$ utilisé pour la normalisation est l'opérateur min (resp. max).

e-SLDD$_\star$

Dans [Fargier *et al.*, 2013c], nous considérons le monoïde $\langle \mathbb{R}^+ \times \mathbb{R}^+, \star, \langle 0, 1\rangle\rangle$ qui permet d'inclure le cadre AADD dans la définition étendue de SLDD.

Soit $E = \mathbb{R}^+ \times \mathbb{R}^+$, $1_s = \langle 0, 1\rangle$, et $\otimes = \star$ définie par $\forall b, b', c, c' \in E$, $\langle b, c\rangle \star \langle b', c'\rangle = \langle b + c \times b', c \times c'\rangle$. $\mathcal{E} = \langle E, \star, 1_s\rangle$ est un monoïde.

Soit l'opérateur $min_\star$ défini par $\forall b, b', c, c' \in E$, $\langle b, c\rangle min_\star \langle b', c'\rangle = \langle min(b, b'), max(b + c, b' + c') - min(b, b')\rangle$. Cet opérateur respecte bien la propriété de commutativité et d'idempotence. De plus, l'opérateur $\star$ est distributif à gauche par rapport à $min_\star$ (voir [Fargier *et al.*, 2013c] pour les preuves).

e-SLDD$_\star$, utilisant le monoïde $\langle \mathbb{R}^+ \times \mathbb{R}^+, \star, \langle 0, 1\rangle\rangle$ et l'opérateur de normalisation $min_\star$ est, à un détail près, l'équivalent du langage AADD dans le cadre e-SLDD.

Le détail réside dans le fait que la fonction d'interprétation d'un AADD $f^{\mathcal{AADD}}$ a pour ensemble d'arrivée $\mathbb{R}^+$ quand la fonction d'interprétation $f^{e-\mathcal{SLDD}_\star}$ d'un e-SLDD$_\star$ a pour ensemble d'arrivée $\mathbb{R}^+ \times \mathbb{R}^+$. Ainsi dans un AADD, tout arc pointant sur un sous-AADD ayant une fonction d'interprétation égale à 0, doit avoir sa valuation c à 0, ce qui n'est pas forcément le cas pour un e-SLDD$_\star$ (voir figure 3.7).

Nous proposons aussi les monoïdes correspondant aux SLDD$_{min}$ et SLDD$_{max}$. Voici donc un ensemble de paires monoïdes / opérateurs de normalisation pouvant être exploitées :

- e-SLDD$_+$: $\mathcal{E} = \langle \mathbb{R}^+ \cup +\infty, +, 0\rangle$, normalisation selon min.
- e-SLDD$_\times$: $\mathcal{E} = \langle \mathbb{R}^+, \times, 1\rangle$, normalisation selon max
- e-SLDD$_\star$: $\mathcal{E} = \langle \mathbb{R}^+ \times \mathbb{R}^+, \star, \langle 0, 1\rangle\rangle$, normalisation selon $min_\star$
- e-SLDD$_{min}$: $\mathcal{E} = \langle \mathbb{R}^+ \cup +\infty, min, +\infty\rangle$, normalisation selon max.
- e-SLDD$_{max}$: $\mathcal{E} = \langle \mathbb{R}^+, max, 0\rangle$, normalisation selon min.

Figure 3.7 – *Différence entre un e-SLDD$_\star$ et un AADD. À gauche un e-SLDD$_\star$ non normalisé, en haut à droite notre e-SLDD$_\star$ normalisé, en bas à droite normalisé façon AADD.*

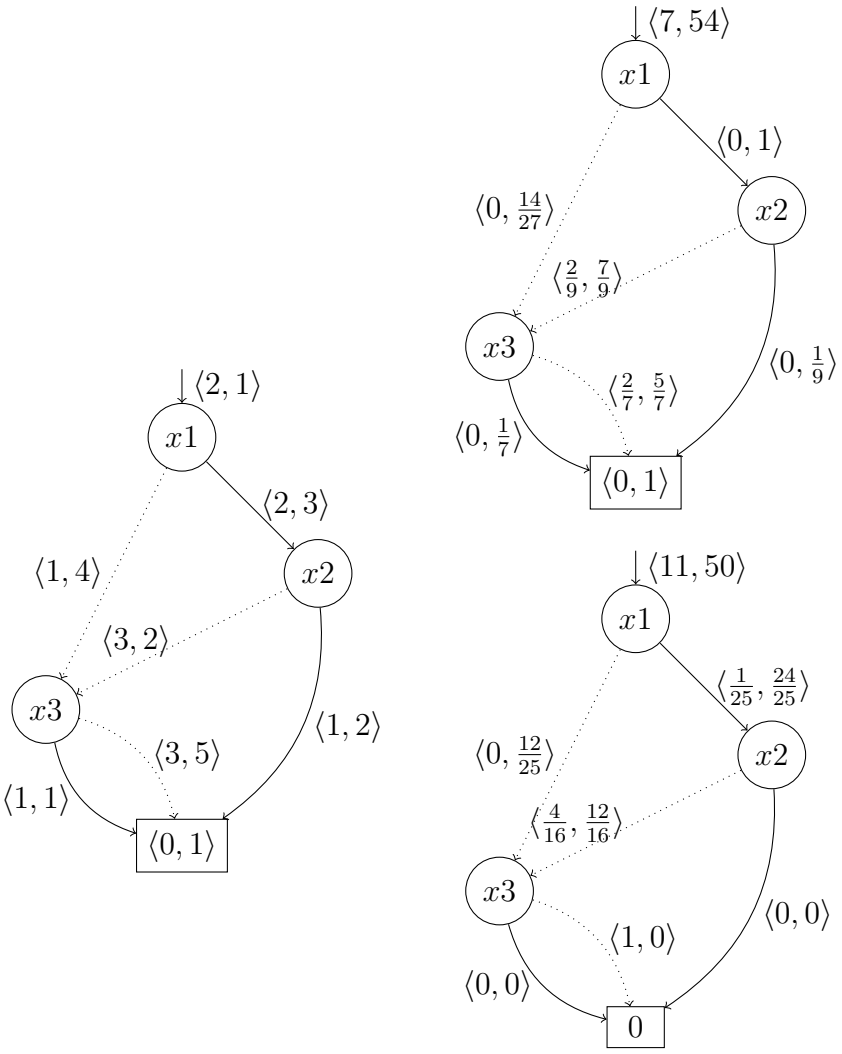


Table de valuation correspondante.

x1	x2	x3	e-SLDD$_\star$	AADD
0	-	0	$\langle 15, 20\rangle$	$\langle 35, 0\rangle = 35$
0	-	1	$\langle 7, 4\rangle$	$\langle 11, 0\rangle = 11$
1	0	0	$\langle 31, 30\rangle$	$\langle 61, 0\rangle = 61$
1	0	1	$\langle 19, 6\rangle$	$\langle 25, 0\rangle = 25$
1	1	-	$\langle 7, 6\rangle$	$\langle 13, 0\rangle = 13$

Figure 3.8 – *Exemple de normalisation d'un nœud d'un SLDD$_+$. À gauche, nœud non normalisé. À droite, nœud normalisé.*

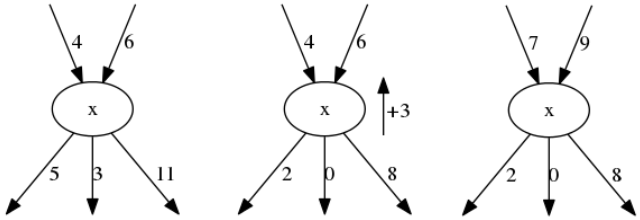
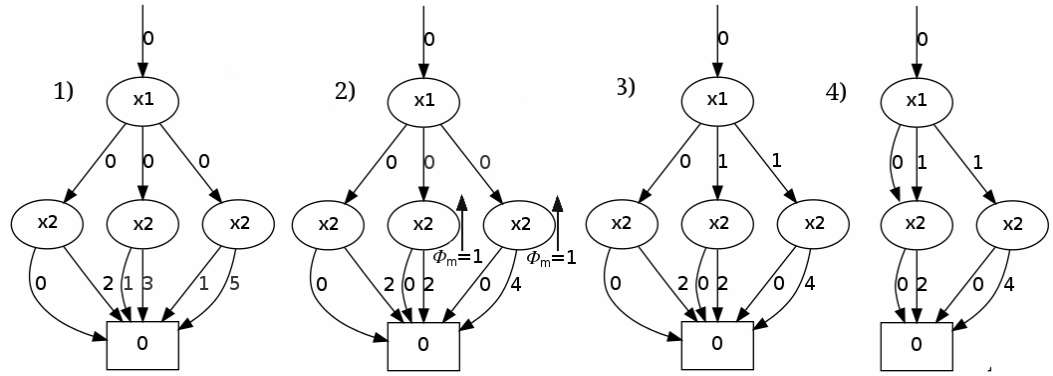


Figure 3.9 – *Normalisation et réduction d'un SLDD$_+$. 1) SLDD non normalisé. 2)-3) Les poids sont remontés sur les arcs supérieurs. 4) Fusion des nœuds isomorphes.*



3.2 Propriétés

3.2.1 Procédure de normalisation

Lors de la construction ou modification d'un SLDD ou AADD, celui-ci n'est pas nécessairement normalisé. Voici donc une procédure permettant de normaliser un nœud.

Cette procédure consiste à remonter les poids sur les arcs sortants vers les arcs entrants, de façon à laisser, suivant l'opérateur $\oplus$ la valuation 1_s. Par exemple pour un SLDD$_+$, on soustrait à tous les arcs sortants la valuation minimale, de façon à avoir 0 sur un des arcs, et on ajoute cette valuation minimale à l'arc ou aux arcs entrants (voir figures 3.8 et 3.9).

Nous proposons ici un algorithme permettant la normalisation d'un nœud d'un SLDD$_+$, SLDD$_\times$, SLDD$_{min}$ et SLDD$_{max}$ (voir algorithme 1).

Ce premier algorithme ne fonctionne pas pour les AADD/e-SLDD$_\star$ à cause de propriétés différentes associées à l'élément neutre de ce langage. Nous pouvons cependant toujours utiliser l'algorithme proposé par Sanner et McAllester [2005] (algorithme 2) pour normaliser de tels diagrammes.

Algorithme 1 : NormaliseNoeudSLDD(N)

input : Un nœud N de SLDD
fonction : Normalise le nœud N
// $\texttt{SLDD}_+$: $\otimes = +, \otimes^{-1} = -, \oplus = min, 1_s = 0$
// $\texttt{SLDD}_\times$: $\otimes = \times, \otimes^{-1} = \div, \oplus = max, 1_s = 1$
// $\texttt{SLDD}_{min}$: $\otimes = min, \otimes^{-1} = max, \oplus = max, 1_s = +\infty$
// $\texttt{SLDD}_{max}$: $\otimes = max, \otimes^{-1} = min, \oplus = min, 1_s = 0$

$\phi_m \leftarrow \oplus_{a \in Out(N)} \phi(a)$;
for *each* $a \in Out(N)$ do
  if $\phi(a) = \phi_m$ then $\phi(a) \leftarrow 1_s$else $\phi(a) \leftarrow \phi(a) \otimes^{-1} \phi_m$
for *each* $a \in In(N)$ do
  $\phi(a) \leftarrow \phi(a) \otimes \phi_m$;

Algorithme 2 : NormaliseNoeudAADD(N)

input : Un nœud N de AADD
fonction : Normalise le nœud N

$b_{min} \leftarrow \min_{a \in Out(N)} b_a$;
$range \leftarrow \max_{a \in Out(N)}(b_a + c_a) \quad b_{min}$;
for *each* $a \in Out(N)$ do
  if $range > 0$ then
    $b_a \leftarrow (b_a - b_{min})/range$;
    $c_a \leftarrow c_a/range$;
  else // Ici, $c_a = 0$ et $b_a = b_{min}$
  $b_a \leftarrow b_a - b_{min}$
for *each* $a \in In(N)$ do
  $b_a \leftarrow b_a + (b_{\min} \times c_a)$;
  $c_a \leftarrow c_a \times range$;

Comme la normalisation d'un nœud modifie la valeur des arcs en amont, cette procédure doit s'appliquer sur l'ensemble des nœuds en commençant logiquement du nœud terminal vers la racine. On peut pour cela parcourir la liste de l'ensemble des nœuds de bas en haut, ou partir de la racine avec une fonction récursive qui traite les fils avant de se normaliser lui même, en notant les nœuds qui ont été normalisés afin de ne pas les traiter plusieurs fois.

La première méthode permet de parcourir l'intégralité des nœuds, quand la 2ème méthode peut aussi être utilisée pour ne normaliser qu'un sous-graphe.

La normalisation d'un graphe permet de mettre en évidence des nœuds isomorphes. Un SLDD ou AADD normalisé et réduit sera donc nécessairement de taille inférieure ou égale à un SLDD ou AADD non normalisé représentant la même fonction.

3.2.2 Compacité théorique

Cette partie vise à comparer la compacité des différents VDD.

La transformation d'un ADD en SLDD_+ ou en $\text{SLDD}_\times$ peut se faire facilement 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 $\text{SLDD}_\times$) les nœuds terminaux sont remplacés par le nœud terminal du nouveau SLDD. Il ne reste alors plus qu'à normaliser et réduire le diagramme obtenu selon les principes de normalisation du SLDD.

La première étape n'augmentera pas la taille du diagramme, et comme énoncé précédemment la normalisation ne peut également que réduire la taille du graphe. Ce qui conduit à la conclusion suivante :

$$\text{SLDD}_+ \leqslant \text{ADD} \text{ et } \text{SLDD}_\times \leqslant \text{ADD}$$

On peut suivre un raisonnement similaire pour la transformation d'un SLDD_+ ou d'un $\text{SLDD}_\times$ en AADD. Tout SLDD_+ peut être transformé en AADD en remplaçant, pour chaque arc a à destination du nœud terminal 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 $\langle \phi(a'), 1 \rangle$, et tout $\text{SLDD}_\times$ peut être transformé en AADD en remplaçant, pour chaque arc a à destination du nœud terminal 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 $\langle\, 0, \phi(a') \rangle$. Il ne reste alors plus qu'à normaliser et réduire le diagramme obtenu selon les principes de normalisation de l'AADD.

La première étape n'a aucune influence sur la taille du diagramme et la normalisation ne peut que réduire la taille du graphe. Ce qui conduit cette fois encore à la conclusion suivante :

$$\text{AADD} \leqslant \text{SLDD}_+ \text{ et } \text{AADD} \leqslant \text{SLDD}_\times$$

Par transitivité de $\leqslant$, nous pouvons également conclure que :

$$\text{AADD} \leqslant \text{ADD}$$

Pour la suite, nous allons introduire deux fonctions portant sur n variables, une première que nous appellerons $plush$ et une deuxième que nous appellerons $foish$, telles que $\forall \vec{x}$, $plush(\vec{x}) = \sum_{i=0}^{n-1} x_i 2^i$ et $\forall \vec{x}$, $foish(\vec{x}) = \prod_{i=0}^{n-1} x_i 2^{2^i}$. Notez le caractère additif de la fonction $plush$ et multiplicatif de la fonction $foish$.

La fonction $plush$ (resp. $foish$), de par sa nature additive (resp. multiplicative) peut être représentée par un SLDD$_+$ (resp. SLDD$\times$) avec seulement $n+1$ nœuds et $2n$ arcs, chaque variable x_i étant représenté par un nœud dont les deux arcs pointent sur le nœud suivant correspondant à la variable x_{i-1}, l'arc étiqueté 1 ayant une valuation de 2^i (resp. 2^{2^i}) et l'arc étiqueté 0 ayant une valuation de 0 (resp. 1). Dans un même temps, cette fonction peut prendre 2^n valuations différentes, soit pour un ADD 2^n nœuds terminaux, et donc un nombre de nœuds total supérieur à 2^n (voir figure 3.10 (resp. figure 3.12)).

Nous sommes donc en présence d'une fonction dont la représentation en SLDD$_+$ est de taille $n+1$ et dont toute représentation en ADD est de taille supérieure à 2^n, ainsi que d'une fonction dont la représentation en SLDD$_\times$ est de taille $n+1$ et dont toute représentation en ADD est de taille supérieure à 2^n. Puisque SLDD$_+ \leqslant$ ADD et SLDD$_\times \leqslant$ ADD, nous pouvons donc conclure :

$$\text{SLDD}_+ < \text{ADD et SLDD}_\times < \text{ADD}$$

Dans [Fargier *et al.*, 2013c] (proposition 10), nous prouvons que la traduction de la fonction $plush$ du langage ADD au langage SLDD$\times$ ne se réduit pas (à l'exception des nœuds terminaux). L'idée est que pour chaque nœud correspondant à la dernière variable, ces nœuds étant au nombre de 2^{n-1}, deux arcs a et a' en sortent avec pour valuation $a = x$ et $a' = x+1$, les x étant tous différents d'un nœud à l'autre. Les opérateurs de la normalisation utilisés étant $\oplus = max$ et $\otimes^{-1} = \div$, nous avons donc $\phi_m = x+1$, ce qui se normalise en $\phi(a) = \frac{x}{x+1}$ et $\phi(a') = 1$. Les valeurs x étant toutes différentes, aucune paire de nœuds isomorphes ne se dégage. La fonction dans le langage SLDD$_\times$ est donc de taille supérieure à 2^{n-1} nœuds alors que exprimée en SLDD$_+$ elle à une taille de $n+1$ nœuds (voir figure 3.11).

De même, toujours dans [Fargier *et al.*, 2013c] (proposition 10), nous prouvons que la traduction de la fonction $foish$ du langage ADD au langage SLDD$_+$ ne se réduit pas (à l'exception des nœuds terminaux). Des 2^{n-1} nœuds correspondants à la dernière variable, deux arcs a et a' en sortent avec pour valuation $a = 2^x$ et $a' = 2^{x+1}$, les x étant tous différents d'un nœud à l'autre.

Figure 3.10 – *Fonction plush avec $n = 3$ exprimée en (a) sous la forme d'un ADD et en (b) sous la forme d'un $SLDD_+$.*

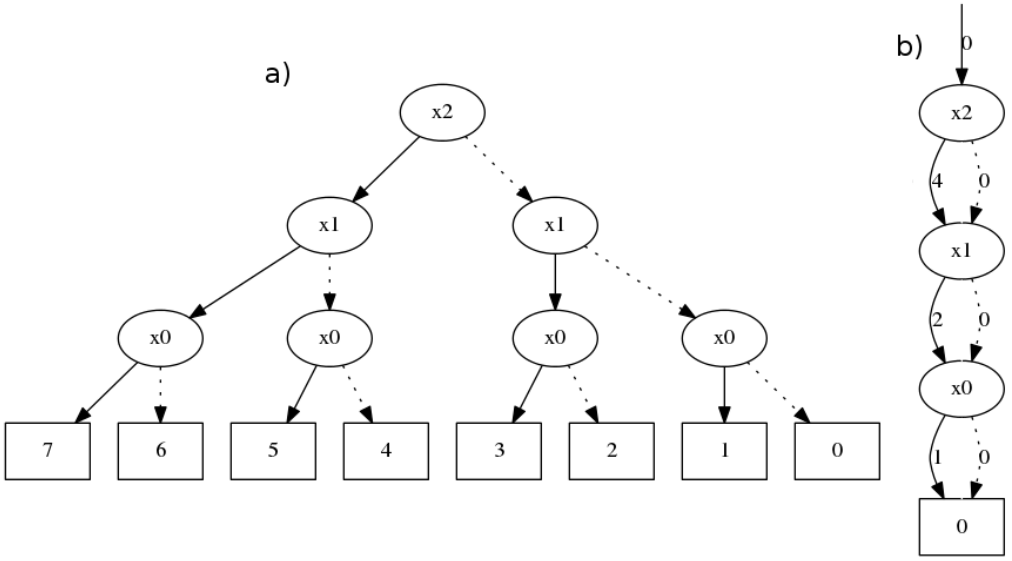


Figure 3.11 – *Fonction plush avec $n = 3$ exprimée en (a) sous la forme d'un ADD et en (b) et (c) sous la forme d'un $SLDD_\times$ non normalisé puis normalisé.*

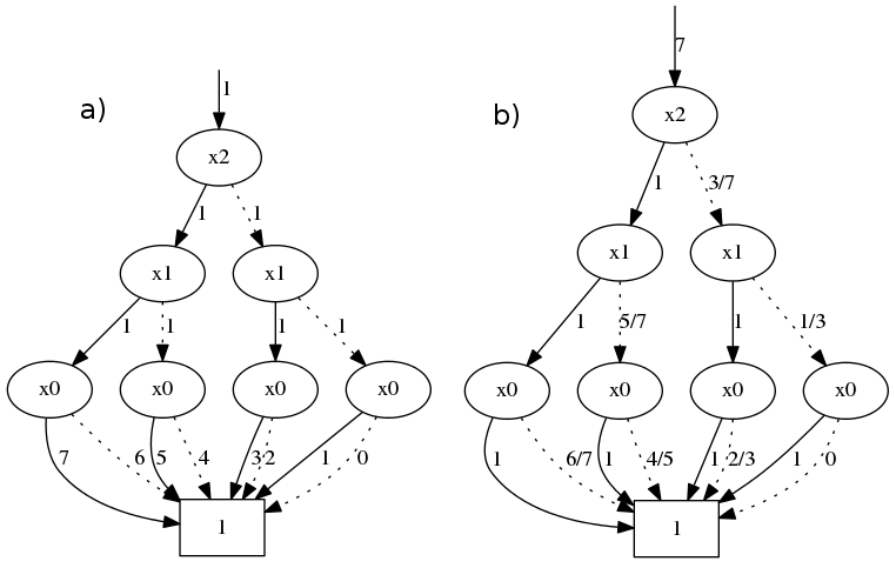


Figure 3.12 – *Fonction foish avec $n = 3$ exprimée sous la forme d'un ADD en (a), d'un SLDD$_\times$ non normalisé en (b) puis normalisé en (c).*

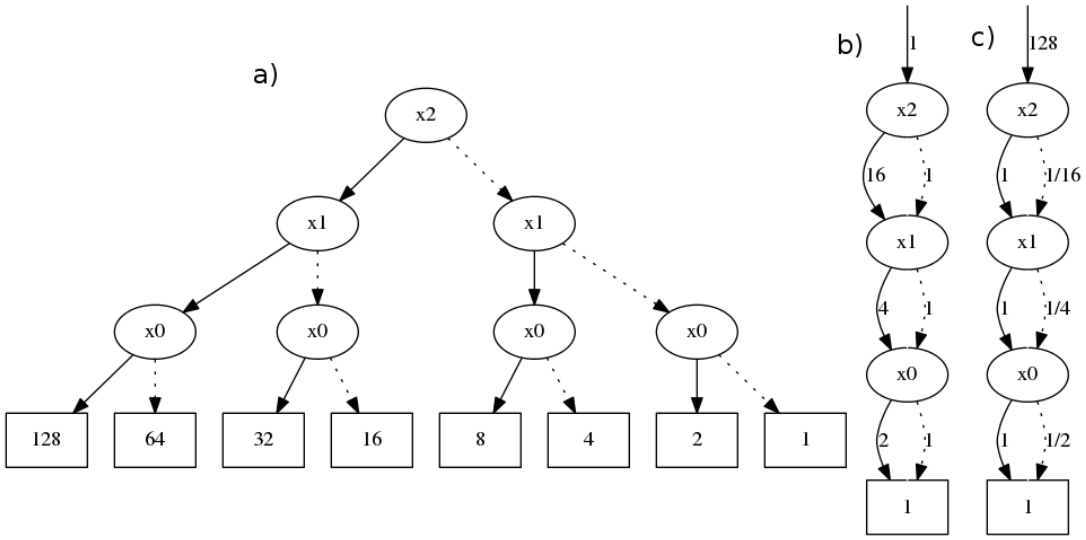


Figure 3.13 – *Fonction foish avec $n = 3$ exprimée sous la forme d'un SLDD$_+$ non normalisé en (a) puis normalisé en(b).*

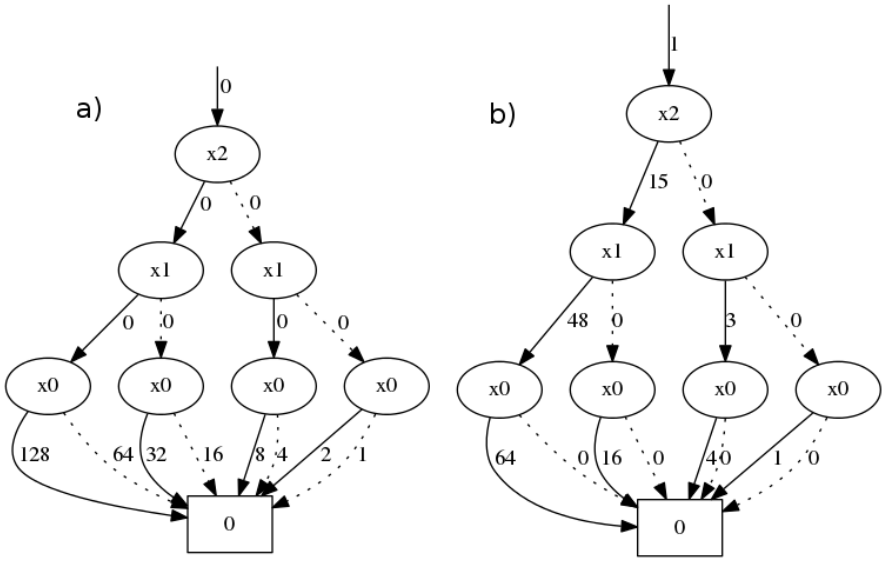
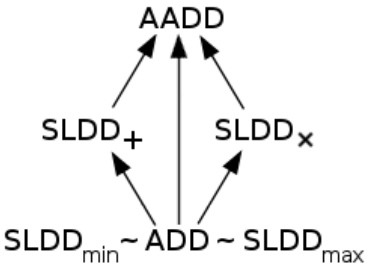


Figure 3.14 – *Hiérarchie de compacité entre les différents langages. $L1 \leftarrow L2$ signifie $L1 < L2$.*



Les opérateurs de la normalisation utilisés étant $\oplus = min$ et $\otimes^{-1} = -$, nous avons donc $\phi_m = 2^x$, ce qui se normalise en $\phi(a) = 0$ et $\phi(a') = 2^{x+1} - 2^x = 2^x$. Les valeurs x étant toutes différentes, aucune paire de nœuds isomorphes ne se dégage. La fonction dans le langage SLDD_+ est donc de taille supérieure à 2^{n-1} nœuds, alors qu'exprimée en $\text{SLDD}_\times$, elle a une taille de $n+1$ nœuds (voir figure 3.13).

Nous pouvons donc conclure des deux paragraphes précédents que :

$$\text{SLDD}_+ \nleqslant \text{SLDD}_\times \text{ et } \text{SLDD}_\times \nleqslant \text{SLDD}_+$$

Sachant que $\text{AADD} \leqslant \text{SLDD}_+$, $\text{AADD} \leqslant \text{SLDD}_\times$, $\text{SLDD}_+ \nleqslant \text{SLDD}_\times$ et $\text{SLDD}_\times \nleqslant \text{SLDD}_+$, nous pouvons conclure :

$$\text{AADD} < \text{SLDD}_+ \text{ et } \text{AADD} < \text{SLDD}_\times$$

Par transitivité nous obtenons :

$$\text{AADD} < \text{ADD}$$

Les résultats précédents sont résumés à la figure 3.14.

Nous avons également montré dans [Fargier *et al.*, 2013c] (proposition 10) que $\text{ADD} \sim \text{SLDD}_{min}$ et $\text{ADD} \sim \text{SLDD}_{max}$, et donc que chacun des langages SLDD_+, $\text{SLDD}_\times$ et AADD sont plus succincts que les langages SLDD_{min} et SLDD_{max}.

3.2.3 D'un langage à un autre

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 sont l'enchaînement d'une transformation permettant de basculer dans la syntaxe d'un autre langage sous une forme non normalisé, suivie d'une normalisation du diagramme ainsi obtenu.

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

La traduction d'un SLDD_+ en AADD se fait en remplaçant, pour chaque arc a à destination du nœud terminal 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 et réduit ensuite l'AADD obtenu, ce qui permet éventuellement d'obtenir une structure de plus petite taille.

De la même façon, tout $\text{SLDD}_\times$ peut être transformé en AADD en remplaçant, pour chaque arc a à destination du nœud terminal 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 $\langle\ 0, \phi(a') \rangle$. On normalise et réduit ensuite l'AADD obtenu, ce qui permet également d'obtenir éventuellement une structure plus compacte que le diagramme original.

Toujours de la même façon, on peut transformer un ADD en SLDD_+ ou en $\text{SLDD}_\times$ 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 $\text{SLDD}_\times$) les nœuds terminaux sont remplacés par le nœud terminal du nouveau SLDD. On normalise et réduit ensuite le diagramme obtenu selon les principes de normalisation des SLDD_+ (resp. des $\text{SLDD}_\times$).

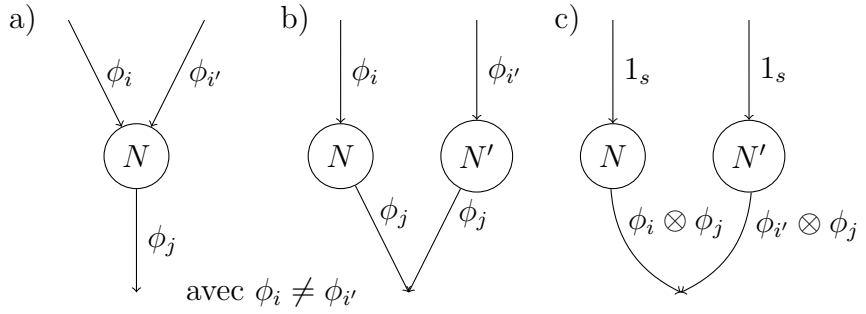
La transformation d'un ADD en AADD, est similaire, les valuations ϕ_i des nœuds terminaux étant reportées sur leurs arcs entrants sous la forme d'un couple $\langle \phi_i, 0 \rangle$, les autres arcs se voyant attribuer la valuation $\langle 0, 1 \rangle$.

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

Si les transformations précédentes permettent une réduction de la taille du diagramme, avec les transformations à venir, nous risquons une augmentation de la taille, voire même une explosion exponentielle de la taille du diagramme d'arrivée.

La première étape peut être vue comme le passage d'un peigne, repoussant vers le bas certaines valuations. Comme un nœud qui peut être séparé en deux cheveux, si deux arcs pointent sur un même nœud, et qu'une valuation différente est descendue sur chacun d'eux, alors le nœud est dupliqué et chaque arc possède alors son propre nœud destinataire (voir figure 3.15).

Figure 3.15 – *Opération de « peignage » d'un SLDD. En a), un extrait d'un SLDD que l'on souhaite transformer en ADD. En b), on duplique le nœud N et en c), on descend les valuations sur les arcs inférieurs.*



Transformer un SLDD en ADD revient à repousser les valuations ϕ 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 nœud terminal en autant de nœuds finaux que de valuations différentes sur ses arcs entrants. La procédure de transformation du SLDD en ADD procède de la racine vers le nœud terminal. Plus précisément, pour tout nœud N dont les parents ont été traités :

- remplacer N par autant de copies N_1, … N_k de N que de valuations différentes $\phi_1, \ldots, \phi_k$ sur les arcs entrant dans N ;
- si N est le nœud terminal du diagramme, chaque copie N_i porte la valuation ϕ_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 $\phi(a') = \phi(a) \otimes \phi_i$

On peut ensuite 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, …, N_k de N que de facteurs multiplicatifs différents c_1, …, c_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 c_i : chaque arc a' porte la même valeur du domaine

de sa variable que a et sa valuation est $\langle b_a, 1\rangle$ 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 $\langle b, c\rangle$, on crée un arc a' de N_i vers M, étiqueté par $val(a)$ et prenant pour valuation $\langle b' \times c_i, c' \times c_i\rangle$.

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 $\langle 0, c\rangle$. Pour cela, on crée autant de copies N_1, ..., N_k de N que de facteurs $b_1 + c_1$, ..., $b_k + c_k$ différents sur les arcs a_j entrant dans N. Chaque copie a' d'un $a \in In(N)$ porte la valuation $\langle 0, c_a\rangle$. 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 $\langle b, c\rangle$, on crée un arc a' de N_i vers M, étiqueté par $val(a)$ et prenant pour valuation $\langle b' + \frac{b_i}{c_i}, c'\rangle$. Finalement, les étiquettes des arcs entrant dans le nœud terminal sont mises sous la forme $\langle 0, b + c\rangle$. On transforme ensuite le AADD en SLDD$_\times$ en ne gardant sur les arcs que le coefficient multiplicatif c, et en donnant au nœud terminal la valuation 1.

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 $b_i + c_i$ différentes sur les arcs entrant dans N et le facteur $b_i + c_i$ est repoussé sur les arcs sortant des N_i : les étiquettes $\langle b', c'\rangle$ des arcs a' deviennent $\langle b_i + c_i \times b', c_i \times c'\rangle$.

Deuxième partie

Compilation, requêtes et transformations

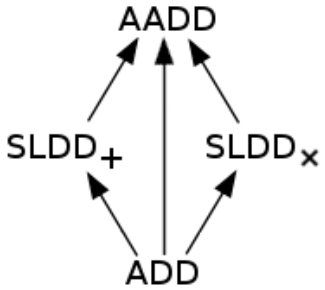
Chapitre 4

Carte de compilation

Nous reprenons dans ce chapitre les résultats publiés dans [Fargier *et al.*, 2014a]. Le but étant ici de poursuivre le travail entamé par Darwiche et Marquis [2002] visant à identifier la complexité de diverses requêtes et transformations sur plusieurs langages, ainsi que la compacité relative de ces langages. Cependant, même si cette carte a été maintes fois élargie, autant pour ce qui concerne les langages étudiés que sur les requêtes et transformations, seul le cas des fonctions booléennes a été étudié, et le cas des VDD n'a été que marginalement abordé.

Nous avons déjà dans les chapitres précédents considéré le sujet de la compacité relative des VDD (rappel figure 4.1). Cependant, si la complexité de certaines requêtes (comme l'optimisation) et de certaines transformations (comme le conditionnement) sur de tels langages est bien connue, il reste de nombreuses requêtes et transformations importantes dont la complexité n'a pas encore été identifiée.

Figure 4.1 – *Hiérarchie de compacité entre les différents langages.* $L1 \leftarrow L2$ *signifie* $L1 < L2$.



Nous listerons dans un premier temps les requêtes et transformations que nous étudierons, celles-ci étant à redéfinir vu la nature valuée de nos langages. Nous presenterons ensuite quelques preuves « types » (voir [Fargier *et al.*, 2014a] pour les autres). Enfin nousdécrirons la carte de compilation ainsi obtenue.

4.1 Requêtes et transformations

Les définitions des requêtes et transformations qui vont suivre sont données dans un cadre général pour les langages de représentation dont les éléments représentent des fonctions à valeurs dans un ensemble E, c'est-à-dire que ces requêtes et transformations sont bien définies même lorsque $E \neq \mathbb{R}^+$.

Soit $\succeq$ un préordre sur E. On note $\sim$ sa partie symétrique et $\succ$ sa partie asymétrique.

Nous avons dans un premier temps défini un ensemble de « coupes » pouvant être vues comme un raffinement de l'ensemble des affectations satisfaisant un critère sur sa valuation. On considère des fonctions f portant sur un ensemble de variables X et prenant leurs valeurs dans E. On a :

- $CUT^{\max}(f) = \{\vec{x} \mid \forall \vec{y}, \neg(f(\vec{y}) \succ f(\vec{x}))\}$;
- $CUT^{\min}(f) = \{\vec{x} \mid \forall \vec{y}, \neg(f(\vec{y}) \prec f(\vec{x}))\}$;
- $CUT^{\succeq\gamma}(f) = \{\vec{x} \mid f(\vec{x}) \succeq \gamma\}$;
- $CUT^{\preceq\gamma}(f) = \{\vec{x} \mid f(\vec{x}) \preceq \gamma\}$;
- $CUT^{\sim\gamma}(f) = \{\vec{x} \mid f(\vec{x}) \sim \gamma\}$;

Par exemple, quand f représente une fonction de coût, $CUT^{\max}(f)$ représente l'ensemble des solutions de valuation (prix) maximal, $CUT^{\min}(f)$ représente l'ensemble des solutions de valuation minimale, et $CUT^{\succeq\gamma}(f)$, $CUT^{\preceq\gamma}(f)$ et $CUT^{\sim\gamma}(f)$ représentent l'ensemble des solutions dont les valuations sont respectivement supérieures ou égales à un prix γ, inférieures ou égales à un prix γ.

Chaque requête/transformation est aussi vue comme une propriété que L satisfait ou pas.

4.1.1 Requêtes

Soit L un langage de représentation sur X à valeurs dans un ensemble E ordonné par une relation $\succeq$.

OPT : un langage L satisfait l'optimisation $\mathbf{OPT}_{\max}$ (resp. $\mathbf{OPT}_{\min}$) ssi il existe un algorithme polynomial associant à toute formule f de L la valeur $\max_{\vec{x}} f(\vec{x})$ (resp. $\min_{\vec{x}} f(\vec{x})$).

EQ : un langage L satisfait l'équivalence **EQ** ssi il existe un algorithme polynomial associant tout couple f, g de formules de L à la valeur 1 si $\forall \vec{x}, f(\vec{x}) = g(\vec{x})$, et à la valeur 0 si non.

SE : un langage L satisfait la requête d'implication de formules **SE** ssi il existe un algorithme polynomial associant tout couple f, g de formules de L à la valeur 1 si $\forall \vec{x}, f(\vec{x}) \succeq g(\vec{x})$, et à la valeur 0 si non.

CO : un langage L satisfait la cohérence partielle supérieure (resp. inférieure, resp. égale) $\mathbf{CO}_{\succeq\gamma}$ (resp. $\mathbf{CO}_{\preceq\gamma}$, resp. $\mathbf{CO}_{\sim\gamma}$) ssi il existe un algorithme polynomial associant tout $\gamma \in E$ et toute formule f de L à la valeur 1 si $\exists \vec{x}, f(\vec{x}) \succeq \gamma$ (resp.$f(\vec{x}) \preceq \gamma$, resp. $f(\vec{x}) \sim \gamma$), et à la valeur 0 si non.

VA : un langage L satisfait la validité partielle supérieure (resp. inférieure, resp. égale) $\mathbf{VA}_{\succeq\gamma}$ (resp. $\mathbf{VA}_{\preceq\gamma}$, resp. $\mathbf{VA}_{\sim\gamma}$) ssi il existe un algorithme polynomial associant tout $\gamma \in E$ et toute formule f de L à la valeur 1 si $\forall \vec{x}, f(\vec{x}) \succeq \gamma$ (resp.$f(\vec{x}) \preceq \gamma$, resp. $f(\vec{x}) \sim \gamma$), et à la valeur 0 si non.

ME : un langage L satisfait l'énumération de max-modèles $\mathbf{ME}_{\max}$ ssi il existe un polynôme p et un algorithme associant à toute formule f de L l'ensemble des éléments de $CUT^{\max}(f)$ en temps $p(|f|, |CUT^{\max}(f)|)$.

Les requêtes $\mathbf{ME}_{\min}$, $\mathbf{ME}_{\succeq\gamma}$, $\mathbf{ME}_{\preceq\gamma}$ et $\mathbf{ME}_{\sim\gamma}$ sont définies de la même façon que $\mathbf{ME}_{\max}$, en utilisant respectivement les ensembles $CUT^{\min}(f)$, $CUT^{\succeq\gamma}(f)$, $CUT^{\preceq\gamma}(f)$ et $CUT^{\sim\gamma}(f)$ à la place de $CUT^{\max}(f)$.

MX : un langage L satisfait l'extraction de max-modèles $\mathbf{MX}_{\max}$ ssi il existe un algorithme polynomial associant à toute formule f de L un élément de $CUT^{\max}(f)$.

Les requêtes $\mathbf{MX}_{\min}$, $\mathbf{MX}_{\succeq\gamma}$, $\mathbf{MX}_{\preceq\gamma}$ et $\mathbf{MX}_{\sim\gamma}$ sont définies de la même façon que $\mathbf{MX}_{\max}$, en utilisant respectivement les ensembles $CUT^{\min}(f)$, $CUT^{\succeq\gamma}(f)$, $CUT^{\preceq\gamma}(f)$ et $CUT^{\sim\gamma}(f)$ à la place de $CUT^{\max}(f)$.

CT : un langage L satisfait le comptage de max-modèles $\mathbf{CT}_{\max}$ ssi il existe un algorithme polynomial associant à toute formule f de L le nombre d'éléments de $CUT^{\max}(f)$.

Les requêtes $\mathbf{CT}_{\min}$, $\mathbf{CT}_{\succeq\gamma}$, $\mathbf{CT}_{\preceq\gamma}$ et $\mathbf{CT}_{\sim\gamma}$ sont définies de la même façon que $\mathbf{CT}_{\max}$, en utilisant respectivement les ensembles $CUT^{\min}(f)$, $CUT^{\succeq\gamma}(f)$, $CUT^{\preceq\gamma}(f)$ et $CUT^{\sim\gamma}(f)$ à la place de $CUT^{\max}(f)$.

L'ensemble de ces requêtes présente un intérêt pour la configuration de produit. Considérons par exemple un produit tel qu'une voiture. L'affectation $\vec{x}$ de l'ensemble des variables X correspond à un modèle de voiture et la valuation $f(\vec{x})$ correspond à son prix (la valuation « $\bot$ » correspond à un modèle de voiture irréalisable, ou du moins non proposé par le constructeur).

On considère $\gamma = 1000 Gallions$[1]. La requête $\mathbf{CO}_{\preceq\gamma}$ permet de savoir s'il existe au moins un modèle de voiture coûtant 1000 *Gallions* ou moins, quand $\mathbf{VA}_{\preceq\gamma}$ permet de verifier qu'il n'existe que des modèles valant 1000 *Gallions* ou moins. La requête $\mathbf{OPT}_{\min}$ permet de connaître le prix du modèle de voiture le moins cher, $\mathbf{MX}_{\min}$ donnera la (une des) configuration(s) du modèle de voiture le moins cher, $\mathbf{ME}_{\preceq\gamma}$ donnera l'ensemble des configurations valant 1000 *Gallions* ou moins, et $\mathbf{CT}_{\sim\gamma}$ donnera le nombre exact de modèles de voiture valant exactement 1000 *Gallions*. Les requêtes **EQ** et **SE** sont utiles pour la comparaison de diagrammes (on considère que ces diagrammes utilisent le même ordre « $\leq$ » sur les variables). La requête **SE** permet de s'assurer qu'une représentation donnera toujours, pour deux configurations identiques, une voiture moins cher qu'une autre représentation. La requête **EQ** présente un intérêt technique, car elle permet de détecter des graphes ou sous-graphes isomorphes.

4.1.2 Transformations

Soit L un langage de représentation sur X à valeurs dans un ensemble E, et $\odot$ un opérateur binaire associatif et commutatif sur E.

Soit $\mathcal{X} \subseteq X$ et $\vec{x} \in D_{\mathcal{X}}$, la notation $f_{|\vec{x}}$, la restriction de f à $\vec{x}$, signifie f privé des variables de $\mathcal{X}$, et tel que $f_{|\vec{x}} = f(\vec{x})$.

CD : un langage L satisfait le conditionnement **CD** ssi il existe un algorithmepolynomial associant à toute formule f de L, tout $\mathcal{X} \subseteq X$, et tout $\vec{x} \in D_{\mathcal{X}}$, une formule appartenant à L et équivalente à $f_{|\vec{x}}$.

1. Pour rappel, $1 Gallion = 17 Mornilles = 493 Noises$.

$\odot$**BC** : un langage L satisfait la $\odot$-combinaison bornée $\odot$**BC** ssi il existe un algorithme polynomial associant à tout couple f, g de formules de L, une formule appartenant à L et équivalente à $f \odot g$.

$\odot$**C** : un langage L satisfait la $\odot$-combinaison $\odot$**C** ssi il existe un algorithme polynomial associant à tout ensemble $\{f_1, \ldots, f_n\}$ de formules de L, une formule appartenant à L et équivalente à $\bigodot_{i=1}^{n} f_i$.

$\odot$**Elim** : un langage L satisfait la $\odot$-élimination de variables $\odot$**Elim** (resp. d'une seule variable, **S**$\odot$**Elim**) ssi il existe un algorithme polynomial associant à toute formule f de L et tout ensemble de variables $\mathcal{X} \subseteq X$ (resp. tout singleton $\mathcal{X} = \{x\} \subseteq X$) une formule appartenant à L et équivalente à $\bigodot_{\vec{x} \in D_{\mathcal{X}}} f_{|\vec{x}}$.

SB$\odot$**Elim** : un langage L satisfait la $\odot$-élimination de variable bornée **SB**$\odot$**Elim** ssi il existe un polynôme p et un algorithme associant à toute formule f de L et toute variable $x \in X$ une formule appartenant à L et équivalente à $\bigodot_{\vec{x} \in D_x} f_{|\vec{x}}$, en temps $p(|f|^{|D_x|})$.

$\odot$**Marg** : un langage L satisfait la $\odot$-marginalisation de variable $\odot$**Marg** ssi il existe un algorithme polynomial associant à toute formule f de L et toute variable $x \in X$ une formule appartenant à L et équivalente à $\bigodot_{\vec{z} \in D_Z} f_{|\vec{z}}$ avec $Z = X \backslash \{x\}$.

CUT : un langage L satisfait la coupe max $\mathbf{CUT}_{\max}$ ssi il existe un algorithme polynomial associant toute formule f de L à la valeur 1 si $\vec{x} \in CUT^{\max}(f)$, et à la valeur 0 si non.

Les transformations coupe min, γ-coupe haute, γ-coupe basse et γ-coupe égale ($\mathbf{CUT}_{\min}$, $\mathbf{CUT}_{\succeq\gamma}$, $\mathbf{CUT}_{\preceq\gamma}$ et $\mathbf{CUT}_{\sim\gamma}$) sont définies de la même façon, en utilisant respectivement les ensembles $CUT^{\min}(f)$, $CUT^{\succeq\gamma}(f)$, $CUT^{\preceq\gamma}(f)$ et $CUT^{\sim\gamma}(f)$.

Dans le cas de la configuration de produit, un utilisateur va devoir faire des choix sur un ensemble de variables de configuration qui lui sont proposées.

La transformation **CD** renvoie un diagramme tenant compte d'une affectation réalisée sur une ou plusieurs variables de X, et donc par exemple de mettre à jour la représentation après un choix fait sur une variable par un utilisateur.

Les transformations d'élimination ($\odot$**Elim**, **S**$\odot$**Elim** et **SB**$\odot$**Elim**) et de marginalisation ($\odot$**Marg**) utilisant les opérateurs *min* ou *max* sont particulièrement intéressantes pour la recommandation de produits. Les *min*-élimination et *max*-élimination permettent d'avoir un diagramme, optimal (selon *min* ou

max) en termes de prix, ignorant une ou un ensemble de variables. Les *min*-marginalisation et *max*-marginalisation associent à chaque alternative d'une variable les valuations correspondant aux solutions optimales (selon *min* ou *max*) les incluant.

Les combinaisons suivant les opérateurs « + » et « × » font partie intégrante de la compilation ascendante. Ces transformations correspondent à l'ajout de contraintes à un diagramme (+**BC** pour les problèmes de nature additive, ×**BC** pour les problèmes de nature multiplicative).

Les coupes (**CUT**) permettent de dépouiller la représentation de l'ensemble des solutions dont la valuation ne correspond pas à nos attentes, par exemple parce que trop chères.

4.2 Preuves

Cette section ne reprend pas l'intégralité des preuves données dans [Fargier *et al.*, 2014a] mais seulement certaines preuves « types ». L'intégralité des preuves, extraites de la version étendue de l'article sus-mentionné, peuvent être trouvées en annexe, section C.

Le but est de prouver qu'une requête ou une transformation est ou n'est pas satisfaite par un langage. Pour prouver qu'une requête est satisfaite, il faut donc donner un algorithme réalisant cette opération en un temps polynomial (ou plus simplement montrer que cette requête découle logiquement d'une autre requête satisfaite).

À l'inverse, prouver qu'une requête ou transformation n'est pas satisfaite peut se faire de deux façons. La première consiste à montrer que répondre à cette requête ou effectuer cette transformation permettrait de résoudre un problème connu. Si ce problème connu est NP-difficile, alors cette requête ne peut être de complexité polynomiale (sauf si $P = NP$). L'autre méthode concernant les transformations uniquement consiste à donner un exemple dans lequel la représentation après transformation est exponentiellement plus grande que la forme initiale. La transformation ne peut pas être garantie en temps polynomial dans un tel cas.

4.2.1 Coupes min et max (polynomiales)

Proposition. *Les langages ADD, SLDD$_+$, SLDD$_\times$ et AADD satisfont* **CUT**$_{\max}$, **CUT**$_{\min}$, **OPT**$_{\max}$, **OPT**$_{\min}$, **ME**$_{\max}$, **ME**$_{\min}$, **MX**$_{\max}$, **MX**$_{\min}$, **CT**$_{\max}$ *et* **CT**$_{\min}$, **CO**$_{\succeq\gamma}$, **CO**$_{\preceq\gamma}$, **VA**$_{\succeq\gamma}$, **VA**$_{\preceq\gamma}$ *et* **VA**$_{\sim\gamma}$

Démonstration. Comme (i) les VDD sont des graphes sans circuit, et (ii) l'opérateur d'agrégation $\otimes$ des valuations ϕ est monotone dans les langages

SLDD_+, $\text{SLDD}_\times$ et AADD, on peut facilement adapter un algorithme de recherche de plus court chemin (polynomial en temps) construisant un diagramme de décision, par exemple un MDD, représentant les affectations optimales (maximales ou minimales). Pour le langage ADD, cette opération est évidente.

Ceci implique directement la satisfaction, pour l'ensemble des langages étudiés, des transformations $\mathbf{CUT}_{\max}$ et $\mathbf{CUT}_{\min}$, mais aussi logiquement des requêtes $\mathbf{OPT}_{\max}$ et $\mathbf{OPT}_{\min}$, et des requêtes associées aux coupes $CUT^{\max}$ et $CUT^{\min}$: $\mathbf{ME}_{\max}$, $\mathbf{ME}_{\min}$, $\mathbf{MX}_{\max}$, $\mathbf{MX}_{\min}$, $\mathbf{CT}_{\max}$ et $\mathbf{CT}_{\min}$.

De même, la connaissance des valeurs maximales et minimales du problème nous donne la réponse aux requêtes $\mathbf{CO}_{\succeq\gamma}$, $\mathbf{CO}_{\preceq\gamma}$, $\mathbf{VA}_{\succeq\gamma}$, $\mathbf{VA}_{\preceq\gamma}$ et $\mathbf{VA}_{\sim\gamma}$.

4.2.2 Combinaison bornée (polynomiale)

Proposition. *$SLDD_+$ satisfait* $+\mathbf{BC}$

Démonstration. Prouvons maintenant que le langage SLDD_+ satisfait la requête de combinaison bornée $+\mathbf{BC}$. Considérons deux formules f et f', toutes deux respectant le même ordre sur les variables, et dont les nœuds bégayants n'ont pas été enlevés (les arcs issus d'un nœud étiqueté par une variable x_i pointeront toujours sur un nœud étiqueté par la variable x_{i+1}). Nous voulons construire la formule $g(\vec{x}) = f(\vec{x}) + f'(\vec{x})$.

Soit une affectation $\vec{x}$ et $p(\vec{x}) = \langle a_1, \ldots, a_n \rangle$ $p'(\vec{x}) = \langle a'_1, \ldots, a'_n \rangle$ les chemins correspondants dans f et f'. On a alors :

$$\begin{aligned} g(\vec{x}) &= (\phi_0 + \phi(a_1) + \cdots + \phi(a_n)) + (\phi'_0 + \phi(a'_1) + \cdots + \phi(a'_n)) \\ &= (\phi_0 + \phi'_0) + (\phi(a_1) + \phi(a'_1)) + \cdots + (\phi(a_n) + \phi(a'_n)) \end{aligned}$$

Nous allons ici faire la somme de ces deux graphes niveau par niveau. Pour toute variable x (ou étage du graphe), pour chaque combinaison possible d'un nœud N_i de f avec un nœud N'_j de f' tel que $Var(N_i) = Var(N'_j) = x$, créons un nœud $N_{i,j}$ dans g tel que $Var(N_{i,j}) = x$.

(i) l'offset de g prend la valeur $\phi_0 + \phi'_0$. Il pointe logiquement sur le nœud $N_{1,1}$ de l'étage x_1. (ii) pour chaque nœud $N_{i,j}$ de variable $Var(N_{i,j}) = x$, pour chaque valeur d du domaine de x, si a_d issu de N_i pointe sur M_i et a'_d issu de N'_j pointe sur M'_j alors ajouter un arc α_d reliant $N_{i,j}$ à $M_{i,j}$ avec $\phi(\alpha_d) = \phi(a_d) + \phi(a'_d)$. (iii) tous les nœuds de g non atteignables sont supprimés (voir l'algorithme 3, et l'exemple de +-combinaison de deux SLDD_+ figure 4.2).

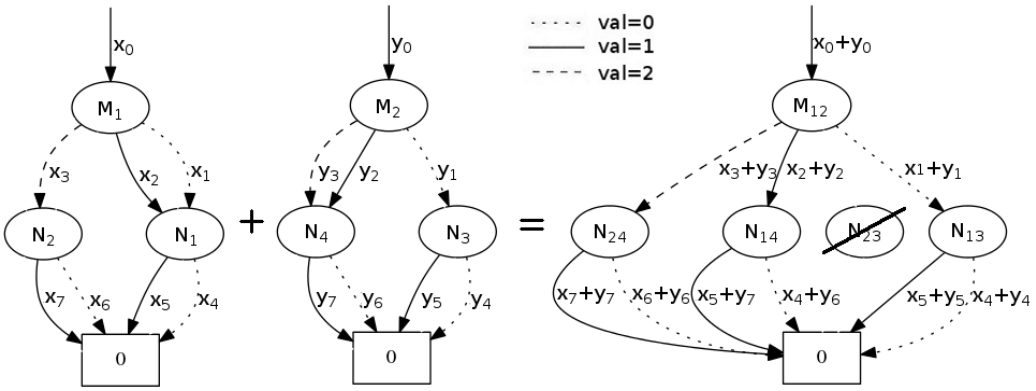
Il en résulte alors une formule g représentant $f + f'$. La formule g est de taille majorée par $|f| \times |f'|$, et au cours de l'exécution, les nœuds de f et f' ne sont parcourus au maximum qu'une seule fois. La transformation $+\mathbf{BC}$ est donc réalisable en temps polynomial sur un SLDD_+.

Algorithme 3 : $\otimes$BC(α, β)

input : α, β 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 α et β
// $NoeudProd(M_\alpha, M_\beta)$ désigne le nœud créé à partir des
// nœuds M_α, M_β

1 $\phi_0(\gamma) \leftarrow \phi_0(\alpha) \otimes \phi_0(\beta)$; // calcul de l'offset de γ
2 $M_\alpha \leftarrow root(\alpha)$;
3 $M_\beta \leftarrow root(\beta)$;
// La source de γ est le produit des sources de α et β
4 Créer un nouveau nœud M_γ avec $Var(M_\gamma) = Var(M_\alpha) = Var(M_\beta)$;
5 $root(\gamma) \leftarrow M_\gamma$;
6 $link_\alpha(M_\gamma) \leftarrow M_\alpha$;
7 $link_\beta(M_\gamma) \leftarrow M_\beta$;
8 $NoeudProd(M_\alpha, M_\beta) \leftarrow M_\gamma$;
9 for *each* $x \in X$, do
10 for *each* M_γ *déjà créé tel que* $Var(M_\gamma) = x$ do
11 for *each* $v \in D_x$ do
12 Ajouter à M_γ un arc sortant a_γ portant la valeur v;
13 $a_\alpha \leftarrow$ arc sortant de $link_\alpha(M_\gamma)$ tel que $val(a_\alpha) = v$;
14 $a_\beta \leftarrow$ arc sortant de $link_\beta(M_\gamma)$ tel que $val(a_\beta) = v$;
// a_γ doit pointer sur le nœud produit des nœuds
// pointés par a_α et a_β
15 if $\nexists\, M'_\gamma$ *tel que* $link_\alpha(M'_\gamma) = Out(a_\alpha)$ *et* $link_\beta(M_\gamma) = Out(a_\beta)$ then
// M'_γ, le produit des nœuds issus de a_α et a_β
// n'a pas encore été créé : on le crée.
16 Créer un nouveau nœud M'_γ;
17 $link_\alpha(M'_\gamma) \leftarrow Out(a_\alpha)$;
18 $link_\beta(M_\gamma) \leftarrow Out(a_\beta)$;
19 $NoeudProd(M_\alpha, M_\beta) \leftarrow M'_\gamma$;
20 $Out(a_\gamma) \leftarrow Prod(M_\alpha, M_\beta)$;
21 $\phi(a_\gamma) \leftarrow \phi(a_\alpha) \otimes \phi(a_\beta)$;
22 NormaliseSLDD$(\gamma, \phi_0(\gamma))$;

Figure 4.2 – *Combinaison additive de deux SLDD$_+$. L'ensemble des nœuds M_i sont étiquetés par la variable $x1$ de domaine 3, l'ensemble des nœuds N_i sont étiquetés par la variable $x2$ de domaine 2, et un nœud N_{ij} est la combinaison des nœuds N_i et N_j.*



Notez qu'une procédure similaire peut permettre de réaliser la transformation $\times$**BC** dans le langage SLDD$_\times$, et toute transformation $\odot$**BC** dans le langage ADD.

4.2.3 Combinaison bornée (non polynomiale)

Proposition. *SLDD$_+$ ne satisfait pas* $\times$**BC**

Démonstration. L'objectif ici est de trouver un exemple pour lequel la combinaison par l'opérateur $\times$ de deux SLDD$_+$ donne un SLDD$_+$ de taille exponentielle dans celle des SLDD$_+$ à combiner. Cette existence prouverait qu'aucun algorithme ne peut garantir la réalisation cette transformation en temps polynomial dans la taille des formules d'entrée.

Prenons deux formules f et f' sur n variables booléennes, telles que $\forall \vec{x}$, $f(\vec{x}) = \sum_{i=0}^{n-1}(x_i \times 2^i)$ et $f'(\vec{x}) = \sum_{i=0}^{n-1}((1 - x_i) \times 2^i)$. Ces formules sont représentées comme montré à la figure 4.3.

Notez que ces fonctions représentent un compteur et un décompteur, et quel que soit $\vec{x}$, on a toujours $f(\vec{x}) + f'(\vec{x}) = 2^n - 1$. Cependant ce n'est pas l'addition mais la multiplication qui nous intéresse. Nous appelons $\vec{x_i}$ l'affectation telle que $f(\vec{x_i}) = i$ et $f'(\vec{x_i}) = 2^n - 1 - i$.

La multiplication des deux SLDD$_+$ peut facilement être exprimée sous la forme d'un ADD, prenant la forme d'un arbre, possédant 2^n nœuds terminaux

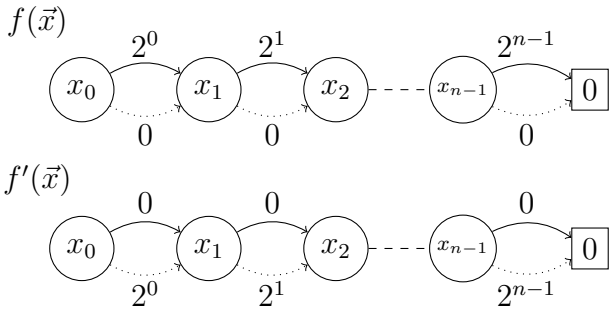


Figure 4.3 – *Représentation en SLDD$_+$ des formules f (en haut) et f' (en bas)*

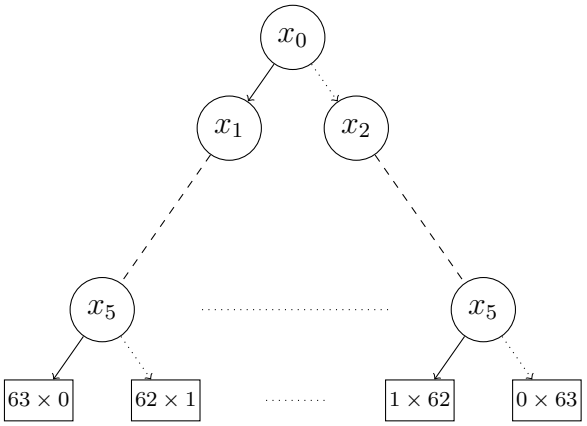


Figure 4.4 – *ADD représentant la combinaison par l'opérateur $\times$ des deux formules f et f', dans le cas où $n = 6$*

portants les valuations $(2^n - 1) \times 0, (2^n - 2) \times 1, \ldots, (2^n - 1 - i) \times i, \ldots, (2^n - 1 - (2^n - 1)) \times (2^n - 1)$ (voir figure 4.4).

Après traduction de cet ADD en SLDD$_+$, les nœuds étiquetés x_{n-1} auront leurs arcs sortants portant comme valuation $\phi(a_1) = (2^n - 1 - i) \times (i)$ et $\phi(a_0) = (2^n - 1 - (i + 1)) \times (i + 1)$ avec i pair. Pour que deux nœuds étiquetés x_{n-1} soient isomorphes, il faudrait que leurs différences respectives $\phi(a_1) - \phi(a_0)$ soient égales. Or :

$$\begin{aligned}\phi(a_1) - \phi(a_0) &= (2^n - 1 - i) \times (i) - (2^n - 1 - (i + 1)) \times (i + 1) \\ &= -i^2 + i(2^n - 1) + i^2 - i(2^n - 2) + i - 2^n + 2) \\ &= -2i - 2^n + 2\end{aligned}$$

Cette différence variant avec la valeur i, il n'y a aucune paire de nœuds étiquetés par la variable x_{n-1} qui soit isomorphes, et donc aucune réduction possible sur le SLDD$_+$. Le nombre de nœuds étiquetés par la variable x_{n-1}

est donc de 2^{n-1} (soit 2^n nœuds dans l'ensemble du SLDD$_+$). Contre $n+1$ nœuds pour les deux SLDD$_+$ en entrée, nous avons ici une explosion en taille lors de cette transformation. On peut en conclure que le langage SLDD$_+$ ne satisfait pas la transformation $\times$**BC**.

Notez qu'un raisonnement similaire permet de montrer que SLDD$_\times$ ne satisfait pas la transformation $+$**BC**. Les mêmes exemples permettent aussi de montrer que le langage AADD ne satisfait ni la transformation $+$**BC**, ni la transformation $\times$**BC**. Cependant la preuve de non-existence de nœuds isomorphes dans le AADD final est plus ardue car la procédure de normalisation l'est également. Tous les détails en annexe, section C, proposition A.30.

4.2.4 Cohérence de la γ-coupe égale (non polynomiale)

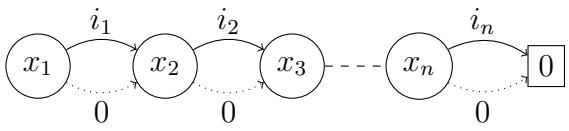
Proposition. *SLDD$_+$ ne satisfait pas* **CO**$_{\sim\gamma}$

Démonstration. Nous allons montrer que le problème *SUBSET SUM* [Garey et Johnson, 1979], problème connu pour être NP-complet, peut être réduit au problème **CO**$_{\sim\gamma}$ sur un SLDD$_+$.

Le problème de la somme de sous-ensembles (*SUBSET SUM*) consiste à déterminer si il existe ou non, parmi un ensemble E de n nombres entiers $\langle n_1, \ldots, n_n \rangle$, un sous-ensemble dont la somme est égale à un entier γ donné.

On peut associer à tout ensemble E, en temps polynomial, un SLDD$_+$ de n nœuds étiqueté par n variables booléennes $\langle x_1, \ldots, x_n \rangle$. Les deux arcs sortant d'un nœud N étiqueté par la variable x_i pointent tous deux vers la variable x_{i+1} avec pour valuation n_i si $x_i = 1$ et 0 si $x_i = 0$ (voir figure 4.5).

Figure 4.5 – *Représentation en SLDD$_+$ d'une instance du problème SUBSET SUM*



Résoudre **CO**$_{\sim\gamma}$ sur un tel SLDD$_+$ reviendrait donc à résoudre le problème *SUBSET SUM*. On peut conclure que le langage SLDD$_+$ ne satisfait pas **CO**$_{\sim\gamma}$ (sauf si $P = NP$).

4.3 Carte de compilation des VDD

Sautons nombres de preuves propositions corollaires et lemmes, et venons-en directement aux résultats, Tables 4.1 et 4.2.

Table 4.1 – *Résultats sur les requêtes de base, l'optimisation et les γ-coupes ; $\surd$ signifie « satisfait » et $\circ$ signifie « ne satisfait pas, sauf si $P = NP$ ». Les résultats pour les problèmes de satisfaction de contraintes valuées additives ($VCSP_+$) sont donnés à titre de comparaison.*

Requête	ADD	$SLDD_+$	$SLDD_\times$	AADD	$VCSP_+$
EQ	$\surd$	$\surd$	$\surd$	$\surd$	?
SE	$\surd$	$\surd$	$\surd$	?	$\circ$
$\mathbf{OPT}_{\max}$ / $\mathbf{OPT}_{\min}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{CT}_{\max}$ / $\mathbf{CT}_{\min}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{ME}_{\max}$ / $\mathbf{ME}_{\min}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{MX}_{\max}$ / $\mathbf{MX}_{\min}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{VA}_{\sim\gamma}$	$\surd$	$\surd$	$\surd$	$\surd$	?
$\mathbf{VA}_{\succeq\gamma}$ / $\mathbf{VA}_{\preceq\gamma}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{CO}_{\sim\gamma}$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$
$\mathbf{CO}_{\succeq\gamma}$ / $\mathbf{CO}_{\preceq\gamma}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{CT}_{\sim\gamma}$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$
$\mathbf{CT}_{\succeq\gamma}$ / $\mathbf{CT}_{\preceq\gamma}$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$
$\mathbf{ME}_{\sim\gamma}$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$
$\mathbf{ME}_{\succeq\gamma}$ / $\mathbf{ME}_{\preceq\gamma}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$
$\mathbf{MX}_{\sim\gamma}$	$\surd$	$\circ$	$\circ$	$\circ$	$\circ$
$\mathbf{MX}_{\succeq\gamma}$ / $\mathbf{MX}_{\preceq\gamma}$	$\surd$	$\surd$	$\surd$	$\surd$	$\circ$

Table 4.2 – *Résultats sur les transformations ; $\surd$ signifie « satisfait » et $\bullet$ signifie « ne satisfait pas ».*

Transformation	ADD	$SLDD_+$	$SLDD_\times$	AADD
CD	$\surd$	$\surd$	$\surd$	$\surd$
$\mathbf{CUT}_{\max}$ / $\mathbf{CUT}_{\min}$	$\surd$	$\surd$	$\surd$	$\surd$
$\mathbf{CUT}_{\sim\gamma}$	$\surd$	$\bullet$	$\bullet$	$\bullet$
$\mathbf{CUT}_{\succeq\gamma}$ / $\mathbf{CUT}_{\preceq\gamma}$	$\surd$	$\bullet$	$\bullet$	$\bullet$
$\max$ **BC** / $\min$ **BC**	$\surd$	$\bullet$	$\bullet$	$\bullet$
$+$**BC**	$\surd$	$\surd$	$\bullet$	$\bullet$
$\times$**BC**	$\surd$	$\bullet$	$\surd$	$\bullet$
$\max$ **C** / $\min$ **C**	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$+$**BC** / $\times$**C**	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$\max$ **Elim** / $\min$ **Elim**	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$+$**Elim** / $\times$**Elim**	$\bullet$	$\bullet$	$\bullet$	$\bullet$
S $\max$ **Elim** / **S** $\min$ **Elim**	$\bullet$	$\bullet$	$\bullet$	$\bullet$
S $+$ **Elim** / **S** $\times$ **Elim**	$\bullet$	$\bullet$	$\bullet$	$\bullet$
SB $\max$ **Elim** / **SB** $\min$ **Elim**	$\surd$	$\bullet$	$\bullet$	$\bullet$
SB $+$ **Elim**	$\surd$	$\surd$	$\bullet$	$\bullet$
SB $\times$ **Elim**	$\surd$	$\bullet$	$\surd$	$\bullet$
$\max$ **Marg** / $\min$ **Marg**	$\surd$	$\surd$	$\surd$	$\surd$
$+$**Marg**	$\surd$	$\surd$	$\surd$	$\surd$
$\times$**Marg**	$\surd$	?	$\surd$	?

4.4 Discussions

Il faut tout d'abord noter que, comme attendu, les VDD se comportent bien face aux requêtes étudiées. En effet, ils satisfont l'ensemble des requêtes portant sur les valuations maximales et minimales. Dans bien des cas ces requêtes sont même de complexité linéaire dans le nombre de variables, voire même constant pour **OPT** ou **CO**, lorsque le VDD est normalisé [2].

Les requêtes basées sur les coupes $CUT^{\succeq\gamma}$ et $CUT^{\preceq\gamma}$, sont également satisfaites dans la majorité des cas, ce qui n'est pas le cas des requêtes associées à la coupe $CUT^{\sim}$. Cela montre qu'il est difficile de répondre aux questions de type « donne moi tous les produits valant exactement 1000 *gallions* » (utile pour quelqu'un souhaitant retrouver sa configuration de la veille dont il ne se souvient que du prix), ou encore « réduis la recherche aux produits coûtant entre x et y » (pour permettre à l'utilisateur de se placer dans une gamme de produits).

L'utilisation du langage ADD pour ce genre de requêtes pourrait bien constituer une solution, cependant l'augmentation en espace qu'elle entraîne peut être dissuasive.

Les transformations essentielles à la configuration de produit sont également satisfaites.

Un point intéressant à souligner est la satisfaction des requêtes $+$**BC** et $\times$**BC** pour respectivement les langages SLDD_+ et $\text{SLDD}_\times$, alors que le langage AADD ne les satisfait pas.

La non-satisfaisabilité de ces requêtes pour le langage AADD rendent la compilation en AADD plus complexe.

2. L'offset d'un SLDD ou d'un AADD contient la valuation optimale, ainsi que l'assurance d'une solution.

Chapitre 5

Compilation

Ce chapitre traite des méthodes utilisées pour la compilation de réseaux de contraintes valuées, pour passer de l'ensemble des contraintes du problème à un diagramme représentant les solutions du problème.

5.1 Préambule

5.1.1 Les contraintes

Avant d'attaquer la méthode de compilation de réseaux de contraintes en VDD, nous allons nous arrêter un peu sur le format des contraintes. Il est important de noter la différence entre les problèmes de nature additive, où l'opérateur d'agrégation des valuations associé aux contraintes est l'opérateur $+$ (valuation portant sur un coût par exemple), et les problèmes de nature multiplicative où l'opérateur d'agrégation des valuations associé aux contraintes est l'opérateur $\times$ (valuation portant sur une probabilité par exemple).

Chaque contrainte c est exprimée sous la forme d'une table T, portant sur un ensemble de variables $Var(T) \subseteq X$, et associant à chaque n-uplet t de T (affectation des variables de T) une valuation $t[\phi]$. On à donc si $t \subseteq \vec{x}$, alors $f_T(\vec{x}) = t[\phi]$.

Une contrainte c est dite « souple » si elle associe une valuation $t[\phi] \in E$, à chaque n-uplet t de T, avec $\mathcal{E} = \langle E, \otimes, \succ \rangle$. On peut éventuellement considerer une valuation par défaut ϕ_0 associée aux affectations non explicitement présentes dans la table T.

Une contrainte c est dite « dure » si elle associe l'élément neutre de $\mathcal{E}$ (0 pour un SLDD_+, 1 pour un $\text{SLDD}_\times$) aux n-uplets qui la satisfont, et l'élément

absorbant de $\mathcal{E}$ ($+\infty$ pour un SLDD$_+$, 0 pour un SLDD$_\times$) aux affectations non explicitement présentes dans la table. Les contraintes dures sont aussi parfois exprimées sous une forme dite de « conflit », où tous les n-uplets se voient attribuer directement l'élément absorbant.

On suppose que les variables de chaque contrainte sont ordonnées selon le même ordre que celui utilisé dans le VDD.

5.1.2 Méthode de compilation

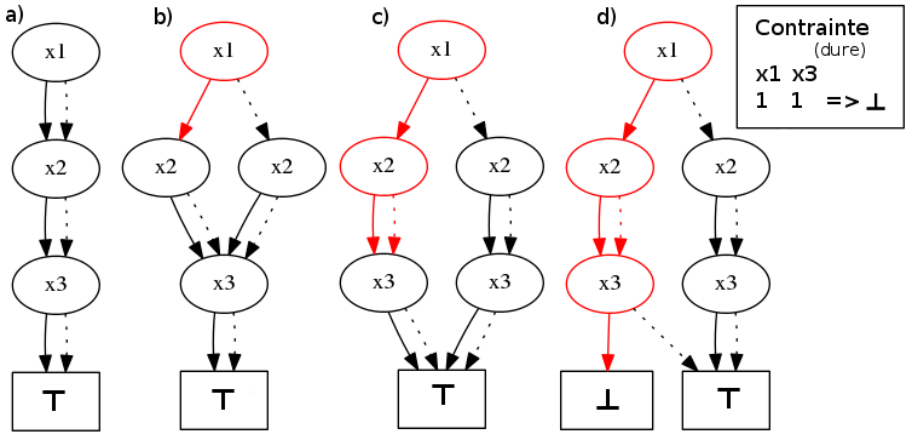
De par la propriété de canonicité pour les les langages décris dans le chapitre précédent, la méthode de compilation utilisée n'a aucune influence sur le résultat, seulement sur le temps de compilation et l'espace mémoire maximal occupé durant la phase de compilation.

Il existe deux différentes approches parmi les méthodes de compilation, les méthodes descendantes (*top-down*) et les méthodes ascendantes (*bottom-up*). Les méthodes descendantes partent d'un ensemble vide de solution et cherchent à ajouter des solutions afin de compléter la représentation, jusqu'à avoir une représentation de l'ensemble des solutions du problème. Les contraintes ne sont pas traitées individuellement mais dans leur ensemble. Afin de ne pas parcourir l'ensemble de solutions, Huang et Darwiche [2004] proposent d'exploiter la trace du solveur (son arbre de recherche), et de détecter les sous-problèmes déjà traités au cours de la compilation. Cette méthode a pour avantage de maintenir une représentation qui reste, tout au long de la construction, de taille inférieure ou égale à la taille de la représentation finale. Elle a cependant pour défaut d'être généralement beaucoup plus lente que les méthodes ascendantes. Ainsi, en théorie, si la capacité mémoire allouée au problème est suffisante pour exprimer le résultat de la compilation et si on leurs en donne le temps, les méthodes descendantes finiront toujours par compiler le problème donné.

À l'inverse, les méthodes ascendantes partent d'une représentation non contrainte, et ajoutent les contraintes une par une à la représentation. Ces méthodes sont généralement plus rapides que les méthodes descendantes. Cependant, ces méthodes étant plus gourmandes en mémoire, certains problèmes ne sont tout simplement pas compilables. En effet, au cours de la compilation, la représentation courante peut être de taille supérieure à la taille de la représentation finale.

C'est pourtant une méthode ascendantes que nous avons choisi d'étudier, alléchés par la promesse d'un gain de temps.

Figure 5.1 – *Exemple d'ajout d'une contrainte $x1 \wedge x3 \Rightarrow \bot$ sur un OBDD de trois variables $x1$, $x2$ et $x3$. En a) un BDD « blanc ». De b) à d) on isole progressivement tous les chemins correspondants à $x1 \wedge x3$ avant de leur affecter l'élément $\bot$ en d).*



5.2 Compilation ascendante

5.2.1 Principe de compilation

Notre approche de compilation d'un réseau de contraintes valuées en VDD suit le procédé ascendant classique pour la construction de diagrammes de décisions ordonnés, valués ou non, proche des procédés proposés par Sanner et McAllester [2005], Bryant [1986], ou Amilhastre [1999]. La différence (mineure) réside dans le fait que, pour les auteurs précédemment cités, chaque contrainte est d'abord compilée sous la forme d'un diagramme de décision avant d'être combinée au diagramme de décision en cours de construction, quand le nôtre combine directement le diagramme de décision en cours aux tables de contraintes.

Nous déterminons tout d'abord un ordre total $<$ des variables selon lequel le diagramme de décision sera ordonné (la section suivante décrit et compare les heuristiques d'ordonnancement de variables). On crée ensuite un diagramme de décision « blanc », c'est-à-dire un diagramme 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 $<$ (voir figure 5.1-a).

Pour chaque contrainte, on considère chaque n-uplet. La méthode consiste alors à « isoler » tous les chemins correspondants à un n-uplet t des autres chemins afin de pouvoir appliquer la valuation à ces chemins uniquement.

5.2.2 Application aux SLDD

Nous commencons par la compilation en SLDD car c'est la plus simple à comprendre. Nous partons d'un SLDD « blanc » et ajoutons les n-uplets au fur et à mesure. Nous expliquons ici la méthode permettant d'ajouter un n-uplet d'une contrainte à la représentation, cependant l'ensemble des n-uplets d'une même table doivent normalement être traités en parallèle. Nous étudions ici la compilation de contraintes additives sur un SLDD_+. Pour le compilation de contraintes multiplicative sur un $\text{SLDD}_\times$, il suffit de remplacer les additions par des multiplications, les occurrences de l'élément neutre 0 par l'élément neutre 1, et les occurrences de l'élément absorbant $+\infty$ par l'élément absorbant 0.

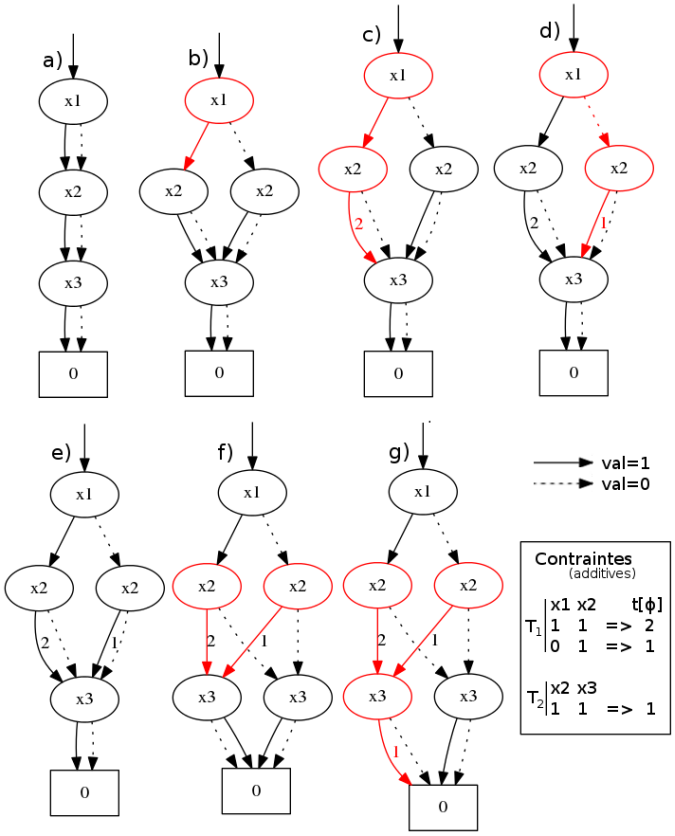
Pour tout nœud étiqueté par la première variable du n-uplet que l'on souhaite ajouter, on sélectionne l'arc correspondant à son affectation dans t, le nœud sur lequel pointait cet arc est alors dupliqué et l'arc pointe désormais seul sur le nouveau nœud. On répète cette opération à partir des nouveaux nœuds créés en sélectionnant l'arc correspondant à l'affectation de cette variable dans t, ou, si cette variable n'est pas dans t, en sélectionnant l'ensemble des arcs issus de ce nœud. Une fois l'ensemble des variables de t parcourues, on ajoute à l'arc sélectionné courant la valuation $t[\phi]$ dans le cas d'une contrainte souple, ou on lui attribue l'élément absorbant dans le cas d'une contrainte dure de « conflit ».

Dans le cas d'une contrainte dure classique (qui n'est pas une contrainte de conflit), ou dans le cas d'une contrainte souple avec une valeur par défaut, il faut affecter l'élément absorbant $+\infty$ ou ajouter la valeur par défaut ϕ_0 à l'ensemble des arcs croisés au cours de ce parcours ne correspondant à aucun n-uplet de la table T, d'où la nécessité de traiter l'ensemble des n-uplets d'une table simultanément.

Il est à noter qu'au cours de l'ajout d'une contrainte à un SLDD, seuls les niveaux du diagramme situés entre la première et la dernière variable de la contrainte sont modifiés. Seule la normalisation qui suit cet ajout peut avoir une influence sur les autres niveaux (au dessus de la première variable et en dessous de la dernière variable), mais elle ne peut entraîner qu'une réduction de taille. Cette caractéristique rend la compilation dans le langage SLDD particulièrement efficace.

5.2.3 Exemple de compilation - SLDD

Figure 5.2 – *Exemple d'ajout de contraintes. a) SLDD « blanc ». b) et c) compilation du premier n-uplet de T_1. d) compilation du deuxième n-uplet de T_1. f) et g) compilation du n-uplet de T_2.*



La figure 5.2 présente la compilation de deux contraintes portant respectivement sur les variables $\{x1, x2\}$ et $\{x2, x3\}$. On part en a) d'un SLDD$_+$ blanc (pour ne pas surcharger la représentation, lorsque la valuation d'un arc est égale à l'élément neutre, 0 pour les SLDD$_+$ ou 1 pour les SLDD$_\times$, elle n'est pas affichée). On parcourt ensuite le chemin correspondant au premier n-uplet de T_1. En partant de $x1$ la première variable de T_1, en b) on isole le chemin correspondant à $x1 = 1$. En c) le poids est ajouté au, et uniquement au, chemin correspondant au premier n-uplet de T_1. En d) on exécute la même opération avec le deuxième n-uplet de T_1 (le chemin correspondant à $x1 = 0$ est déjà isolé). Le résultat de la compilation de T_1 est donné en e).

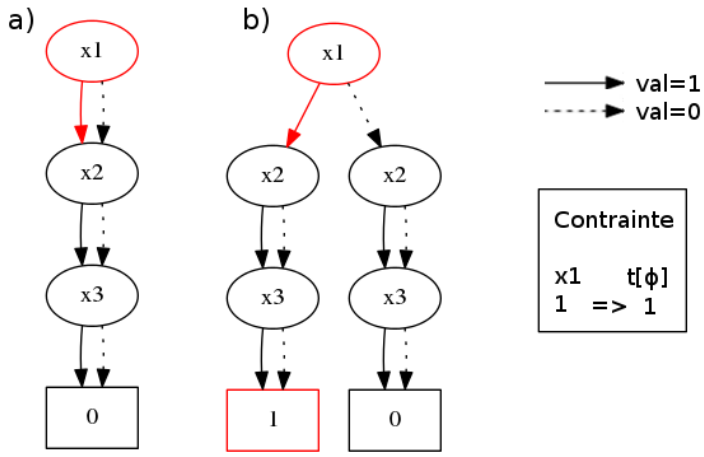
Deux nœuds correspondent à la première variable de la contrainte T_2. On sélectionne donc ces deux nœuds en f) et isole les chemins correspondants à $x2 = 1$. Enfin, on ajoute la valuation en g).

5.2.4 Application aux ADD

L'ajout d'une contrainte sur un ADD suit le même déroulement que pour un SLDD. La différence intervient lorsque l'on souhaite ajouter la valuation $t[\phi]$ (ou toute valuation par défaut ϕ_0). En effet, on ne peut pas ajouter cette valeur sur l'arc correspondant. Il faut donc dupliquer l'intégralité du sous-ADD pointé par cet arc pour ajouter ou multiplier cette valeur aux valuations associées aux nœuds terminaux correspondants. Ainsi, même les contraintes les plus simples, si elles concernent les variables de début de notre ordre, peuvent augmenter considérablement la taille finale de l'ADD.

5.2.5 Exemple de compilation - ADD

Figure 5.3 – *Exemple d'ajout d'une contrainte simple sur un ADD.*



La figure 5.3 montre la compilation d'une contrainte simple portant sur la variable $x1$. En a) le nœud correspondant au chemin $x1 = 1$ est sélectionné. Cependant la valuation ne peut pas être ajouté directement à l'arc, mais seulement à un nœud terminal. Il faut donc dupliquer l'intégralité du sous-graphe afin d'ajouter la valuation à ce chemin uniquement.

5.2.6 Application aux AADD (ou la méthode de l'accordéon)

Dans le cas de l'ajout d'une contrainte de nature additive, la procédure est la même que sur un SLDD uniquement lorsque l'ensemble des facteurs multiplicatifs c des chemins correspondants sont égaux à 1. Si ce n'est pas le cas, la valuation ne peut pas être ajoutée directement sur l'arc car il faut tenir compte du produit des facteurs multiplicatifs en amont de cet arc.

Ainsi, lors de la compilation de contraintes de nature additive en AADD, il faut normaliser la structure après l'ajout de chaque contrainte (sinon le diagramme n'est pas sous sa forme réduite, et la canonicité de la représentation n'est pas garantie) et « dénormaliser » avant l'ajout de chaque nouvelle contrainte. Ainsi, se rapprochant du principe de l'accordéon, tout gain de place lié à la structure plus complexe de l'AADD est systématiquement perdu dans la foulée, puis éventuellement regagné pour être à nouveau perdu, si bien qu'il est difficile de compiler en AADD sans avoir un air de bal traditionnel en tête [1].

Dans le cas d'ajout de contrainte de nature multiplicative, c'est la même musique, chaque valuation devant être ajoutée au facteur multiplicatif c de l'arc correspondant, mais aussi à tous les facteurs additifs b en amont du chemin correspondant, cassant encore un fois tout gain d'espace potentiellement acquis au prix de calculs plus complexes liés à la structure du langage AADD.

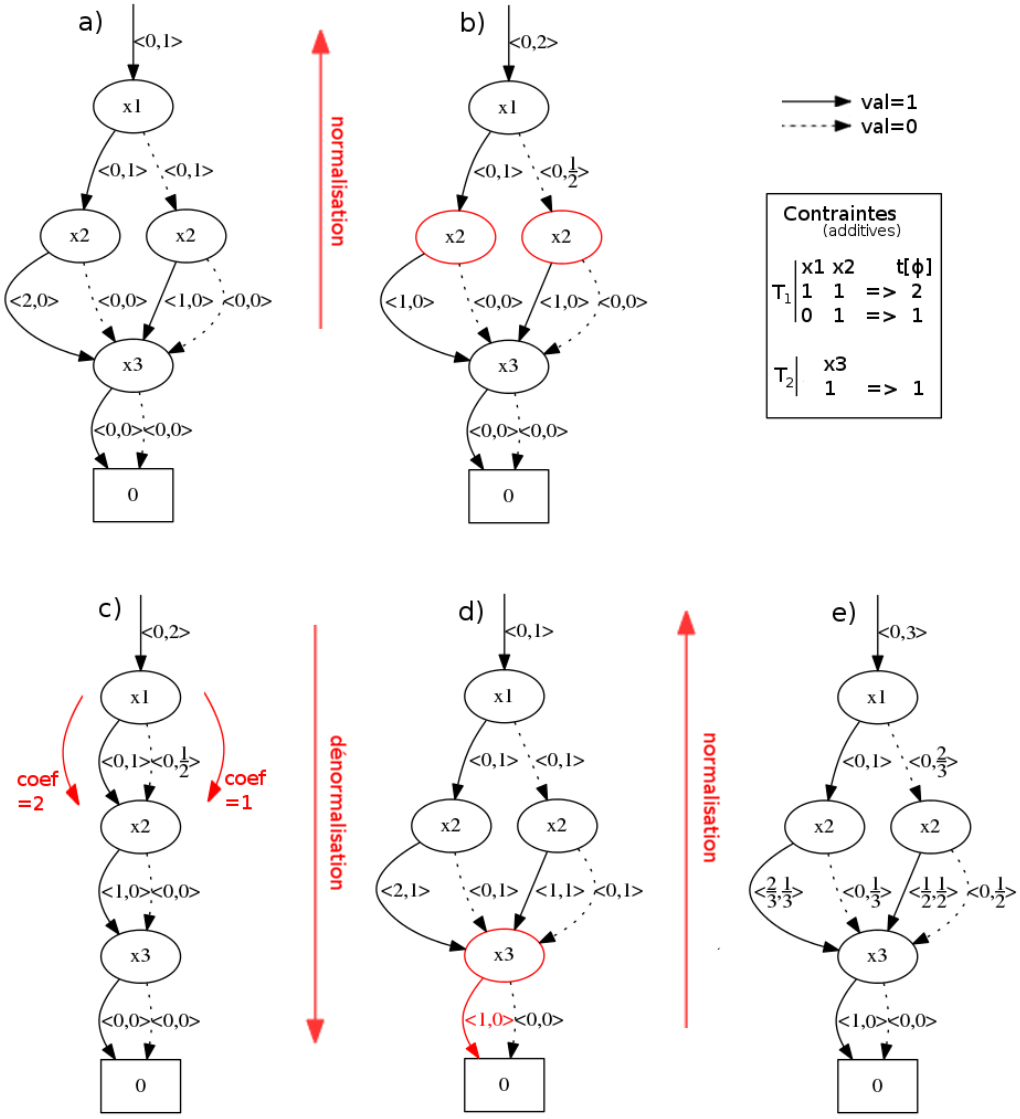
Ce biais est la conséquence directe de la non satisfaction des combinaisons $+\mathbf{BC}$ et $\times\mathbf{BC}$ par le langage AADD.

5.2.7 Exemple de compilation - AADD

La figure 5.4 présente la compilation de deux contraintes portant respectivement sur les variables $\{x1, x2\}$ et $\{x3\}$. La figure a) représente le AADD après l'ajout de la contrainte T_1 (la procédure étant la même que pour un SLDD_+, nous ne l'avons pas répétée) mais avant la procédure de normalisation. Le diagramme est normalisé en b) et réduit en c), ce qui nous permet d'avoir une représentation plus compacte qu'avec le langage SLDD_+. Cependant on ne peut plus ajouter directement les n-uplets de la contrainte T_2 à l'arc de valeur 1 issu de $x3$ tel qu'on l'aurait fait sur un SLDD_+. En effet, suivant le chemin emprunté pour arriver au nœud $x3$, le coefficient multiplicatif n'est pas le même. Il faut alors dénormaliser en d), en faisant exactement l'opération inverse de celle effectuée en b) et c). Après dénormalisation, la valuation peut être ajoutée, et l'AADD renormalisé.

1. D'autant plus quand le compilateur est codé en Java...

Figure 5.4 – *Exemple d'ajout de contraintes sur un AADD. En a) un AADD pas encore normalisé auquel on vient d'ajouter la contrainte T_1. Normalisation en b) et réduction en c) de cet AADD. En d) on « dénormalise » l'AADD pour pouvoir ajouter la contrainte T_2, et on le renormalise en e).*



5.2.8 Le choix du chef

Pour récapituler, la compilation incrémentale en ADD entraîne de très grosses augmentations de taille de la forme compilée, même lors de l'ajout de contraintes simples. La compilation en AADD effectue beaucoup d'opérations de normalisation et dénormalisation inutiles. La compilation en SLDD ne possède aucun défaut, comparée à celles ciblant directement les deux autres langages. En effet, sa structure purement additive ou purement multiplicative s'accorde parfaitement avec le système, purement additif ou purement multiplicatif également, d'agrégation des valuations liés aux contraintes dans les réseaux de contraintes.

Notons également que même si il faut utiliser deux langages distincts, à savoir les SLDD_+ et les $\text{SLDD}_\times$, pour représenter les problèmes de nature respectivement additive et multiplicative, une seule procédure de compilation suffit à ces deux langages. Cette procédure utilise directement les opérateurs $\otimes$, $\otimes^{-1}$ et $\oplus$, eux-mêmes redéfinis par structures de valuation. Le langage SLDD n'est bien qu'un seul langage pouvant faire appel à plusieurs structures de valuation distinctes.

Devant cette domination sans conteste de la compilation SLDD, nous n'avons développé que celle-ci.

5.2.9 Algorithme de compilation

Les algorithmes 4 et 5 décrivent comment incorporer les contraintes données par une table T de n-uplets à un SLDD α. L'algorithme 4 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 α 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 dans [Sanner et McAllester, 2005] pour la compilation en AADD.

Il s'agit de combiner selon $\otimes$ un $\text{SLDD}_\otimes$ α et une fonction associant une valeur $t[\phi]$ à chaque n-uplet t de T (ou la valeur ϕ_0 sinon) en un $\text{SLDD}_\otimes$. Comme chez Bryant [1986], les arcs des deux structures sont suivis simultanément. 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.

Algorithme 4 : AjouteContrainte(T, α, ϕ_0)

input : 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 $f_\alpha \otimes f_T$

```
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);
6 NormaliseSLDD();
```

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 6) 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*.

Pour ne pas alourdir l'algorithme, nous n'avons pas detaillé les actions de normalisation (la fonction NormaliserNoeud(M)). La (re)normalisation du SLDD construit est réalisée « à la volée », c'est-à-dire que dès qu'un nœud M a été traité, 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, ils le seront au retour de la procédure récursive).

S'il existe un nœud isomorphe à M (fonction ExisteNoeudIsomorphe(M)), alors M est fusionné avec lui (lignes 5 et 5), M est alors supprimé, mais pas sans avoir préalablement mémorisé le nœud avec lequel il a été fusionné dans *TableRedirection*. Ainsi si la table *VisitePrecedente* renvoie ultérieurement le nœud M, on saura où le retrouver.

Algorithme 5 : AjouteContrainteArc(T, a, ϕ_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 $f_\alpha \otimes f_T$

// $\otimes = \times$ pour un $\text{SLDD}_\times$; $\otimes = +$ pour un SLDD_+

// si aucun n-uplet dans T

1 if $T = \varnothing$ then

2 $\quad \phi(a) \leftarrow \phi(a) \otimes \phi_0$;

3 // si l'ensemble des variables de $Var(T)$ a été parcouru

4 else if $Var(In(a)) = last(Var(T))$ then

// ici, il ne reste plus qu'un seul n-uplet t dans T

5 $\quad \phi(a) \leftarrow \phi(a) \otimes t[\phi]$;

6 else

7 $\quad M \leftarrow$ Visite(T, a);

// rediriger l'arc a sur M

8 $\quad Out(a) \leftarrow M$;

9 $\quad$ if $Var(M) \in Var(T)$ then

10 $\quad\quad$ for *all arcs* $a' \in Out(M)$ do

11 $\quad\quad\quad T' \leftarrow \{n\text{-uplets } t \text{ de } T \text{ tel que } t[Var(M)] = val(a')\}$;

12 $\quad\quad\quad$ AjouteContrainteArc(T', a', ϕ_0);

13 $\quad$ else

14 $\quad\quad$ for *all arcs* $a' \in Out(M)$ do

15 $\quad\quad\quad$ AjouteContrainteArc(T, a', ϕ_0);

16 $\quad$ NormaliseNoeudSLDD(M);

17 $\quad$ if ExisteNoeudIsomorphe(M) then

18 $\quad\quad M' \leftarrow$ GetNoeudIsomorphe(M);

19 $\quad\quad TableRedirection(M) \leftarrow M'$;

20 $\quad\quad$ FusionNoeuds(M', M);

21

Algorithme 6 : 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
1  M ← Out(a);
   // Le nœud M n'a jamais été atteint lors du traitement de T
2  if VisitePrecedente(M,T) = null then
       // si a est l'unique arc entrant de M
3      if In(M) = {a} then
4          M' ← M;
5      else
6          M' ← Copie(M);
       // Mémoriser M' comme étant le résultat de la visite de
       // M étant donné T
7      VisitePrecedente(M,T) ← M';
8  else
9      M' ← VisitePrecedente(M,T);
10     if TableRedirection(M') ≠ null then
           // si le nœud M' a été fusioné avec un autre nœud, il
           // n'existe plus
11         M' ← TableRedirection(M')
12 
13 return M';
```

Les algorithmes 5 et 6 ne mentionnent pas que si l'algorithme 6 renvoie un nœud déjà traité (c'est-à-dire si on a exécuté la ligne 6, mais pas la ligne 6 de l'algorithme 6), alors le nœud résultant M (algorithme 5 ligne 5) n'a pas besoin d'être traité à nouveau, et on peut n'exécuter que la ligne 5. Cependant si ce même nœud M a été normalisé au cours de l'ajout de cette contrainte, la valuation qui a été remontée au cours de celle-ci aux arcs entrants de M doit également être remontée sur l'arc courant a.

À la fin du traitement AjouteContrainte, on normalise le SLDD selon la procédure NormaliseSLDD() qui parcourt l'ensemble des nœuds du diagramme en leur appliquant l'algorithme 1. On peut même simplifier cette procédure en ne parcourant que les nœuds situés au dessus des nœuds étiquetés $first(Var(T))$.

5.3 Heuristiques

La forte influence de l'ordonnancement des variables sur la taille d'un diagramme de décision a déjà été soulevée par Bryant [1986], Amilhastre [1999], ou Drechsler [2002] dans le domaine booléen, et elle reste bien sûr importante appliquée aux VDD. C'est pourquoi nous avons étudié plusieurs heuristiques, dont certaines utilisées dans les MDD.

Si aucune référence n'est mentionnée, c'est que l'heuristique d'ordonnancement est de nous.

5.3.1 Heuristiques d'ordonnancement de variables

L'objectif de ces heuristiques est de regrouper au maximum les variables intervenant dans une même contrainte. Nous verrons pour cela plusieurs méthodes jouant sur plusieurs critères que l'on cherche à minimiser.

MCF

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 :

$$MCF(x) = |c|, \text{ avec } x \in Var(c) \text{ et } c \text{ est une contrainte dure.}$$

Band-Width [Amilhastre, 1999]

Amilhastre [1999] propose une heuristique basée sur la *Band-Width* du graphe de contraintes. La *Band-Width* d'un ordre correspond à l'écart maximal qu'il peut y avoir dans cet ordre entre deux variables intervenant dans une même contrainte. Plus formellement, avec $O : \{1, \ldots, n\} \mapsto X$ un ordre sur les n variables de X (O associe à chaque rang une variable). La *Band-Width* 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 qui intervient dans le plus de contraintes. La variable suivante est la variable qui, si elle était ajoutée maintenant, maximiserait la *Band-Width* de l'ordre courant (l'idée étant que plus on attendra pour ajouter cette valeure, plus la *Band-Width* de cette variable augmentera. Il s'agit en quelque sortes de la variable qu'il est le plus « urgent » d'ajouter à l'ordre). Plus formellement, 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 :

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

Nous départagerons les ex aequo avec le critère de la variable qui intervient dans le plus de contraintes (MCF), puis en choisissant au hasard si besoin.

MCS [Tarjan et Yannakakis, 1984]

MCS (pour *Maximum Cardinal Search*) est une méthode introduite dans le cadre de reconnaissance de graphes triangulés. Elle permet aux variables fortement contraintes d'être proches des variables avec lesquelles elles sont liées. On choisit les variables de l'ordre itérativement. La première sélectionnée est la variable la plus contrainte. La variable suivante est la variable qui est impliquée dans au moins une contrainte avec le plus de variables déjà sélectionnées. Plus formellement, 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 :

$$MCS_O(x) = |O[i]|, \text{ avec } O[i] \text{ voisine de } x \text{ et } i \text{ variant de 1 à } k.$$

MCS+1

MCS+1 se veut être une amélioration (non gloutonne) de l'heuristique MCS. Le critère $MCS_O(x)$ donne pour chaque variable x une valeur correspondant à la nécessité d'ajouter au plus vite cette variable à l'ordre final (cette valeur ne pouvant qu'augmenter si l'on retarde l'ajout de cette variable à l'ordre final). MCS+1 n'ajoutera pas nécessairement à l'itération k la variable dont la valeur de « nécessité d'un ajout rapide » est la plus forte, mais

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 $y \in Var(c)$.

celle qui minimisera cette valeur en moyenne (sur l'ensemble des variables qui restent à ajouter) à l'itération $k+1$.

Plus formellement, étant donnée une suite O de k variables sélectionnées et $O+x$ la suite O à laquelle on a ajouté la variable x, la variable suivante (au rang $k+1$) est la variable x non encore sélectionnée qui minimise la quantité :

$$\text{MCS+1}_O(x) = \Sigma\{MCS_{O+x}(y)\}.$$

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 $c \in C$:

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

Le centre de gravité d'une contrainte c dépend de la position de chacune des variables $Var(c)$ de 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 :

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

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.

Cette méthode conduit toujours à un point fixe. Cependant, l'ordre obtenu varie selon l'ordre de départ O utilisé.

Comparaison

Ces différentes heuristiques ont été implémentées dans notre compilateur (ainsi que plusieurs autres qui n'ont pas retenu notre attention). Le résultat des expérimentations, dans lesquelles nous comparons ces heuristiques sur les critères de temps de compilation et de taille du diagramme après compilation, est donné dans un chapitre à suivre.

5.3.2 Heuristiques d'ordonnancement des contraintes

Dans le cadre d'une compilation ascendante, les contraintes sont ajoutées une à une au diagramme de décision courant. Ainsi, la taille et la forme du diagramme de décision lors de l'ajout d'une contrainte quelconque dépend des contraintes qui ont été précédemment ajoutées.

L'heuristique d'ordonnancement des contraintes joue un rôle moins déterminant que l'heuristique d'ordonnancement des variables car l'ordonnancement des contraintes n'a aucune influence sur la forme finale compilée d'un problème. Cependant, tout comme l'ordonnancement des variables, l'ordonnancement des contraintes influe fortement sur le temps de compilation, ainsi que sur la taille maximale atteinte par le diagramme durant la compilation, et de ce fait, sur la faisabilité de la compilation d'un problème.

Le problème d'ordonnancement des contraintes est difficile à appréhender car les critères sur lesquels doivent être basées les heuristiques ne sont pas clairs. En effet, s'il est évident qu'il faut rapprocher autant que possible les variables intervenant dans une même contrainte lors de l'ordonnancement des variables, ce qu'il faut faire des contraintes est moins évident. Souhaite-t-on compiler d'abord les contraintes les plus « difficiles », au risque de ralentir la compilation des contraintes plus « faciles » par la suite à cause de l'importante augmentation de la taille du diagramme courant après compilation de ces contraintes, ou au contraire souhaite-t-on compiler d'abord les contraintes les plus « faciles », au risque de ralentir la compilation déjà laborieuse des contraintes les plus « difficiles » à cause d'une faible, mais lourde de conséquence, augmentation de la taille du diagramme ?

L'ordonnancement des contraintes est logiquement effectué après l'ordonnancement des variables car l'ordre des contraintes n'a aucune influence sur l'ordre des variables alors que l'ordre des variables a une influence sur l'ordre des contraintes.

Ordre naturel ou ordre naturel inversé

Bien que très contestable, cette méthode n'est pas nécessairement si stupide que cela. Elle consiste à garder inchangé (ou éventuellement inversé) l'ordre sur les contraintes donné par le fichier encodant le réseau bayésien. En effet, dans de nombreux cas cet ordre n'a pas été choisi au hasard, mais correspond à une réalité physique, ou de conception du produit. Plus formellement (si cela est bien nécessaire), avec $O_c : \{1, \ldots, n\} \mapsto C$ un ordre sur les n contraintes de C (O_c associe à chaque rang une contrainte), et O_{c_i} l'ordre initial :

$$O_c(k) = O_{c_i}(k).$$
ou
$$O_c(k) = O_{c_i}(n - k + 1)$$

BCF

La méthode BCF (pour *Biggest Constraint First*) est une méthode basique qui met en tête les contraintes portant sur le plus de variables. Les contraintes sont donc d'autant plus prioritaires que la valeur BCF est haute :

$$BCF(c) = |Var(c)|.$$

Dureté d'une contrainte

Cette méthode consiste à compiler les contraintes les plus dures en priorité. La dureté d'une contrainte correspond à la proportion de n-uplets non autorisés par rapport au nombre de n-uplets possibles au vue des variables impliquées. Ce sont a priori les contraintes qui réduiront le plus rapidement le nombre de modèles de notre diagramme de décision. Un score de dureté correspondant au rapport n-uplets non autorisés / n-uplets possibles est calculé, et les contraintes sont ensuite triées de la plus dure à la moins dure, les contraintes « souples » étant considéré comme moins dures que l'ensemble des contraintes « dures ». Plus formellement, avec T_c la table de n-uplets de la contrainte c, les contraintes sont triées de la plus petite à la plus grande valeur de :

$$Dureté(c) = \frac{|T_c|}{\Sigma_{x \in Var(c)}(|dom(x)|)}.$$

Difficulté de compilation d'une contrainte

Cette méthode consiste à évaluer la difficulté de compilation d'une contrainte, et ainsi de traiter les plus difficiles le plus tôt possible. Cette évaluation est basée sur le nombre de variables impliquées dans une contrainte, sur la taille des domaines de chacune des variables, ainsi que sur la « *Band-Width* » de cette contrainte, c'est-à-dire « l'écart » entre la première et la dernière variable impliqué dans cette contrainte suivant l'ordre des variables.

Cependant, l'alchimie parfaite entre tous ces paramètres est toujours en cours de recherche. Nous présenterons tout de même par la suite les résultats d'une heuristique triant les contraintes par ordre décroissant suivant le critère :

$$Difficulté(c) = max_{x,y \in Var(c)|x \neq y}(|dom(x)| \times |dom(y)) \times BW(c)$$

Hard first

Il ne s'agit pas d'une heuristique mais d'une modification pouvant être apportée à chacune des heuristiques précédentes (excepté *Dureté d'une contrainte* qui l'incorpore déjà). Il s'agit de compiler en priorité les contraintes dites « dures » et de finir par les contraintes dites « souples ». En effet dans de nombreux problèmes industriels, les contraintes dures constituent la « trame » ou le « squelette » du diagramme. Dans ces cas-là, la taille du diagramme après compilation des contraintes dures varie peu lors de l'ajout des contraintes souples. Il est donc préférable de compiler ce « squelette » avant d'y ajouter les valuations.

Comparaison

Comme les heuristiques d'ordonnancement de variables, ces heuristiques seront testées et discutées dans un chapitre à suivre.

5.4 Implémentation objet

Cette thèse aura également été l'occasion de développer un compilateur de VDD. Ce compilateur, que nous avons appelé compilateur SALADD (pour *Semi-ring or Affin LAbelled Décision Diagramms*), offre les fonctionnalités suivantes : compilation ascendante de réseau de contraintes vers $SLDD_+$ ou de réseau bayésien vers $SLDD_\times$, traduction d'un problème d'un langage VDD à un autre et réalisation de requêtes et transformations sur ces VDD. Nous allons donc ici, sans pour autant rentrer dans les lignes de code, détailler l'implémentation de ce compilateur.

Ce compilateur a été implémenté en Java, cette section décrit la démarche que nous avons suivie, et valable pour tout langage objet.

5.4.1 Classes

Nous avons découpé notre compilateur en différentes classes. En voici une vue globale, ainsi qu'un diagramme UML en figure 5.5.

Structure - Sadd - Splus - Stimes - Saadd

La classe *Structure* correspond à la structure de valuation du VDD. Ainsi 4 classes héritent de *Structure*, correspondant aux structures de valuation du ADD, SLDD_+, $\text{SLDD}_\times$ et AADD, classe implémentant les opérateurs $\otimes$, $\otimes^{-1}$ et $\oplus$. Une instance de cette classe est associée à chaque arc du VDD. La structure associée au ADD tient plus de la décoration, les arcs des ADD n'étant pas valués.

Arc

Arc est la classe correspondant aux arcs de nos VDD. Elle possède en attribut un nœud père $In(a)$, un nœud fils $Out(a)$, une valuation $\phi(a)$, une valeur $val(a)$ et une variable « $\perp$ » permettant d'indiquer que tout chemin l'incluant est incohérent. Chaque arc peut être construit en lui attribuant un nœud père et un nœud fils (ou seulement un nœud fils dans le cas de l'arc correspondant à l'offset) ou à partir d'un autre arc et d'un nouveau père dans le cas de duplication d'un nœud.

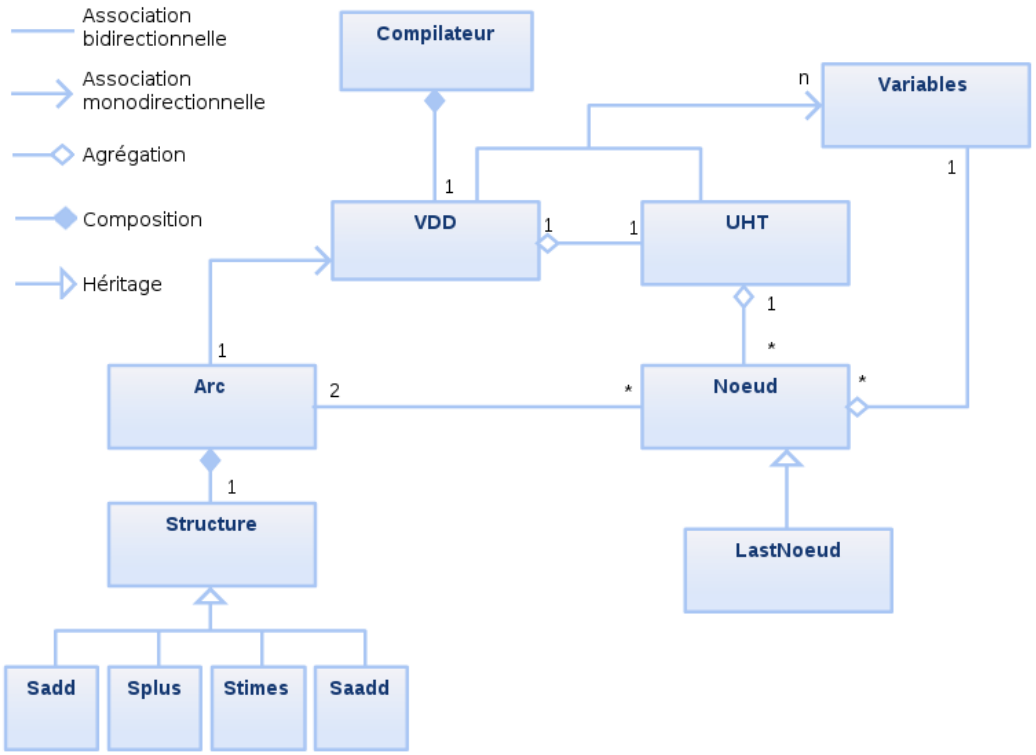
Variable

Cette classe regroupe des informations telles que la taille du domaine Dx de la variable ainsi que sa position dans l'ordre des variables dans le diagramme.

Node

Cette classe possède en attribut la variable associée à chaque nœud, un nombre indéterminé d'arcs entrants, un nombre déterminé (la taille du domaine de la variable associée) d'arcs sortants. Elle possède des méthodes lui permettant de s'auto normaliser, de se fusionner avec un autre nœud isomorphe (un des nœuds est supprimé et l'autre reçoit l'ensemble de ses arcs entrants), ou encore de se dupliquer (le nouveau nœud possède les mêmes arcs sortants, ce qui les rend momentanément isomorphes, et l'un des arc entrant du nœud dupliqué est redirigé vers le nouveau nœud).

Figure 5.5 – *Diagramme UML de classe allégé.*



NodeLast

Cette classe correspond à l'unique nœud terminal d'un VDD. Pour le cas des multiples nœuds terminaux du langage ADD, nous trichons en n'ayant qu'un seul nœud terminal, mais en associant à chaque arc entrant une valuation. Les fonctions de calcul de taille corrigent bien évidement ce biais.

UHT

La classe *UHT* pour *Unique HashTable* référence l'ensemble des nœuds d'un VDD. Ceux ci sont triés par niveau (par variable) et stockés dans une table de hachage dont la clé est calculée en fonction des valuations et fils des arcs sortants, ainsi deux nœuds isomorphes ont toujours la même clé, ce qui facilite les recherches. Cette classe implémente l'ensemble des fonctions touchant à un ensemble de nœuds et ne fonctionnant pas récursivement. On y trouve donc des fonctions de normalisation, de fusion ou de duplication de nœuds, de calcul de taille, de requêtes diverses (voir section 5.4.3 pour plus de détails).

VDD

Cette classe définit ce qu'est notre VDD. Elle contient donc en attribut le premier arc (étiqueté par l'offset) pointant sur le premier nœud, le dernier nœud, la UHT, l'ensemble des variables, la nature du VDD (ADD, SLDD_{+}, $\text{SLDD}_{\times}$ ou AADD), l'état de la normalisation, ... On y trouve un ensemble de fonctions récursives tel que la compilation ou le comptage de solutions, ainsi que l'initialisation de quasiment toutes les opérations.

SALADD

C'est la classe qui gère l'interaction avec l'extérieur : les contraintes qu'il faut compiler et les transformations et requêtes à effectuer. C'est la classe avec laquelle communique l'utilisateur.

5.4.2 Élément absorbant et solution non recevable

Pour des facilités de programmation, nous n'utilisons pas d'élément absorbant dans la structure de valuation, mais ajoutons une variable booléenne dans les arcs correspondant à l'élément « $\perp$ ».

Théoriquement, la sortie d'un arc valué « $\perp$ » devrai être redirigée vers un unique nœud terminal. Cependant, si tous les arcs portant la valuation « $\perp$ » étaient redirigés sur le nœud terminal, le nœud terminal pourrait contenir un nombre trop important d'arcs entrants, ce qui peut entraîner de très fort ralentissements lors des différents traitements. Il est donc conseillé de ne faire pointer ces arcs sur rien.

La descendance de l'arc nouvellement valué « $\perp$ » doit être contrôlée et éventuellement supprimée afin de ne pas avoir un pan de diagramme non accessible.

5.4.3 *Unique HashTable* et fonction de hachage

Nous utilisons une table de hachage pour le stockage de nos nœuds. Plus exactement, nous utilisons autant de tables de hachage que de variables dans le problème. Ainsi, que ce soit pour rechercher des nœud isomorphes, ou même pour effectuer toute opération nécessitant de chercher un nœud en particulier, le temps de recherche ne dépend plus de la taille du diagramme. Le nombre de nœud pouvant croître exponentiellement suivant les fonctions à représenter, la vitesse de recherche, surtout dans les problèmes de grande taille, s'en trouve grandement accélérée.

La clé d'un nœud, ou valeur de hachage, est calculée en s'appuyant sur la définition des nœuds isomorphes, c'est-à-dire d'après la valuation portée et le fils pointé par chacun des arcs (les nœuds d'une même table de hachage ayant tous la même variable...). La valeur de hachage d'un arc est obtenue en additionnant l'identifiant du fils (identifiant unique du nœud) et un entier correspondant à la valuation. La valeur de hachage d'un nœud est obtenue en additionnant toutes les valeurs de hachage des arcs sortants tout en effectuant un décalage circulaire[3] de trois bits entre chaque addition (trois bits car il faut seulement éviter les multiples de deux, le décaclage peut se faire indifféremment vers la droite ou vers la gauche). Une égalité non désirée de deux valeurs de hachage n'est pas préjudiciable car la fonction *equals* vérifie toujours si deux nœuds à valeur de hachage égale sont bien isomorphes.

Plus formellement, cela donne, avec $\#o$ la valeur de hachage d'un objet o, x^i un décalage circulaire de i bits sur la valeur x et $Id(N)$ l'identifiant du nœud N :

$$\#a = \phi(a) + Id(Out(a))$$
$$\#N = \sum_{i=1}^{D_x} \{(\#a_i)^{3i}\} \mid x = Var(N) \text{ et } a_i = Out_i(N)$$

$$equals(N, N') = true \text{ ssi } \forall i, \phi(a_i) = \phi(a'_i) \text{ et } Id(Out(a_i)) = Id(Out(a'_i)),$$
$$\text{avec } a_i = Out_i(N) \text{ et } a'_i = Out_i(N')$$

Deux nœuds sont donc déclarés isomorphes et sont fusionnés ssi il y a collision entre leurs clés et si la fonction *equals* renvoie la valeur « *true* ».

L'utilisation d'une table de hachage induit cependant de nombreuses complications dans l'implémentation. En effet, elle oblige par exemple à retirer de la table de hachage notre nœud avant toute modification sur lui-même ou sur les arcs, de retirer les nœuds parents avant toute modification des arcs entrants sous peine de les perdre à tout jamais car associés à une clé obsolète. Elle complique le parcours de la table de nœuds car ceux-ci changent de valeur de hachage, et donc de position dans la table, au fur et à mesure des modifications apportées aux nœuds. Elle oblige à une parfaite gestion de l'ajout et du retrait des nœuds au cours d'une transformation sous peine de voir fusionner un nœud déjà traité avec un nœud non traité...

3. Le décalage circulaire étant une opération consistant à effectuer un décalage des bits encodant une valuation vers la droite (resp. gauche) avec report des bits de poids faible (resp. poids fort)

5.4.4 Normalisation

Lors du processus de normalisation, le nœud courant doit être retiré de la table de hachage afin de pouvoir être modifié, ainsi que ses nœuds parents afin de modifier également les arcs entrants. Si l'ensemble des nœuds sont valués « $\perp$ », alors la valuation « $\perp$ » doit être remontée sur les arcs entrants et le nœud courant doit être supprimé. De même, si deux nœuds isomorphes sont détectés, la fusion s'opère en redirigeant la sortie des arcs entrants d'un nœud sur l'autre nœud. La clé de hachage des nœuds parents doit également être recalculée.

5.4.5 Arrondis

Certains langages tels que le langage ADD ou, de par sa nature additive, le langage SLDD_+, ne sont pas soumis aux erreurs d'arrondis. Ce n'est cependant pas le cas pour le langage $\text{SLDD}_\times$ avec l'utilisation des opérateurs $\otimes = \times$ et $\otimes^{-1} = \div$, et à plus forte mesure, pour le langage AADD avec la double utilisation des opérateur $+$ et $\times$ ainsi que $-$ et $\div$ lors de la normalisation.

Sanner et McAllester [2005], dans leurs utilisations des AADD, ont choisi de fixer une valeur seuil absolu (e^{-9}) en dessous de laquelle un écart n'est pas considéré comme suffisant et les valeurs sont considérées comme égales. Cette démarche ne permet pas de comparer des valeurs infinitésimales, qui sont pourtant fréquentes quand on compile des réseaux bayésiens. Même si cette approche peut être pertinente dans certains cas, nous souhaitons travailler avec des valeurs exactes et utilisons donc un seuil relatif aux grandeurs comparés.

L'utilisation des classes de structures nous permet également de basculer en système de fraction. Ces fractions sont codées sur des entiers (*long*), voir sur des *bigDecimal* lorsque les *long* ne sont plus suffisants, ce qui arrive assez vite. D'une façon surprenante, l'utilisation de fractions peut dans certains cas améliorer la rapidité comparée à l'utilisation de nombres réels (*double*). Cela est probablement dû à l'opération de division, se résumant à la multiplication des numérateurs et dénominateurs dans le cas des fractions.

5.4.6 Méthodes de transformation

Les transformations ADD→SLDD→AADD sont aisées puisqu'elle ne consistent qu'en un changement de système de valuation et une normalisation. Les transformations AADD→SLDD→ADD nécessitent en plus une étape de « peignage » décrite à la section 3.2.3.

Les transformations $\text{SLDD}_+ \rightarrow \text{SLDD}_\times$ et $\text{SLDD}_\times \rightarrow \text{SLDD}_+$, transformations par ailleurs complètement contre intuitives, demandent quant à elles un passage par la forme ADD ou AADD. Dans nos expérimentations, comme nous nous intéressons davantage à l'exactitude des résultats qu'au temps de transformation, nous préférons réaliser ces transformations en passant par le langage ADD.

De plus si cette transformation est rendue impossible par une explosion en taille du diagramme lors du passage par le langage ADD, on peut supposer qu'une explosion similaire se produirait lors du passage du langage AADD au langage SLDD final désiré.

5.4.7 Module d'implémentation d'heuristique

Le compilateur SALADD que nous avons développé autorise un utilisateur averti à coder sa propre heuristique d'ordonnancement des contraintes ou des variables. Nous mettons à disposition une classe de type *interface* qui peut être implémentée. Le développeur peut acceder à des informations sur les variables (taille du domaine, position dans l'ordre), et sur les contraintes (variables concernées, nombre de n-uplets), ou même utilisé des outils tels que le tableau de contingence des variables, afin de coder sa propre heuristique d'ordonnancement.

Troisième partie

Experimentations et applications à la configuration de produits

Chapitre 6

Expérimentations

Dans une première section, nous allons comparer expérimentalement, d'une part, l'efficacité des différentes heuristiques d'ordonnancement de variables et de contraintes et, d'autre part, la compacité pratique des différents types de VDD étudiés dans les chapitres précédents. La deuxième section sera, quant à elle, consacrée à l'exploitation de la forme compilée. Nous y étudierons différentes requêtes ainsi qu'un protocole de test s'approchant du comportement d'un potentiel utilisateur. Les performances de notre compilateur SALADD seront également comparées à celles d'autres solveurs développés au cours du projet ANR BR4CP.

6.1 Compilation

L'intégralité des expérimentations dont la présentation va suivre a été réalisée grâce à notre compilateur SALADD. Nous utiliserons les abréviations *t-o* et *m-o* pour respectivement *time-out* (>30min) 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.

Pour faire simple, la taille d'un diagramme de décision sera exprimée en nombre de nœuds uniquement, le nombre d'arcs étant étroitement lié à celui-ci.

6.1.1 Jeux d'essai

Nous avons testé nos structures de données et heuristiques sur deux familles de jeux d'essai.

La première famille 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 techniquement faisables, et de contraintes valuées, représentant un coût. Le prix d'un véhicule est ainsi la somme des coûts spécifiés par les différentes contraintes valuées.

Les valuations et noms des variables ont été anonymisés par Renault. 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 dures=147 ; #contraintes valuées=13
- **Medium** : #variables=148 ; taille du domaine max=20 ; #contraintes dures=174 ; #contraintes valuées=24
- **Big** : #variables=268 ; taille du domaine max=324 ; #contraintes dures=332 ; #contraintes valuées=95

Les problèmes **Small Price Only**, **Medium Price Only** et **Big Price Only** sont constitués des seules contraintes de coût (les contraintes « dures » sont omises) des instances **Small**, **Medium** et **Big** respectivement, et les problèmes **Small Hard Only**, **Medium Hard Only** et **Big Hard Only** sont constitués des seules contraintes dures (les contraintes de coût sont omises) des instances **Small**, **Medium** et **Big** respectivement [1].

La seconde famille est un jeu d'essai standard contenant des réseaux bayésiens [Cozman, 2002]. Nous utiliserons quatre instances dont les caractéristiques sont les suivantes :

- **asia** : #variables=8 ; taille du domaine max=2 ; #contraintes valuées=8 (aucune contrainte « dure »)
- **car-starts** : #variables=18 ; taille du domaine max=3 ; #contraintes valuées=16 ; #contraintes dures=6
- **alarm** : #variables=37 ; taille du domaine max=4 ; #contraintes valuées=37 (aucune contrainte « dure »)
- **hailfinder25** : #variables=56 ; taille du domaine max=11 ; #contraintes valuées=49 ; #contraintes dures=20

Notez que si les jeux d'essai de type réseau bayésien sont de natures variées, les trois jeux d'essai de type VCSP additif représentent tous trois le même type

1. Ces jeux d'essai peuvent être téléchargés depuis la page : http://www.irit.fr/~Helene.Fargier/BR4CP/benches.html

de produit issu d'un même constructeur. Si ces jeux d'essai sont du véritable pain béni car ils nous permettent de travailler sur des problèmes réels, le manque de diversité ne nous permet pas de garantir que tout autre problème se comportera de la même façon que les instances fournies par Renault.

6.1.2 Efficacité des heuristiques

Heuristiques d'ordonnancement de variables

Nous testons l'efficacité des différentes heuristiques d'ordonnancement de variables sur deux critères.

Le premier, que l'on peut considérer comme étant le plus important, est la taille du diagramme (nous ne mentionnerons ici que le nombre de nœuds). La taille d'une instance dans un langage donné ne dépend que de l'ordonnancement des variables.

Le deuxième est le temps de compilation. Cette valeur dépend également de l'heuristique d'ordonnancement de contraintes utilisée. Afin de toujours utiliser le même ordre sur les contraintes quelle que soit l'heuristique d'ordonnancement de variables choisie, nous utiliserons « l'ordre naturel », c'est-à-dire l'ordre (pas si) arbitraire (que cela) dans lequel les contraintes apparaissent dans le fichier lu.

Les problèmes de configuration avec fonction de coût ont été compilés en SLDD_{+}, les instances de réseaux bayésiens ont été compilées en $\mathrm{SLDD}_{\times}$. En plus des résultats pour les cinq heuristiques présentées à la section 5.3, nous donnons pour chaque instance les résultats d'ordonnancements aléatoires des variables. 100 tests (100 ordonnancements aléatoires) ont été considérés. Sont relevés le nombre minimal de nœuds obtenus, le nombre moyen de nœuds obtenu, ainsi que le pourcentage de compilations réussies (un échec de compilation pouvant être dû soit à un *time-out* soit à un *out of memory*).

Les résultats sont présentés en table 6.1.

Notons pour commencer que les résultats obtenus en utilisant des ordonnancements produits par les heuristiques considérées sont toujours meilleurs en termes de taille (à une exception près) que ceux produits par ordonnancement aléatoire en moyenne. C'est déjà un bon début.

Table 6.1 – *Comparaison des heuristiques MCF, Band-Width, MCS, MCS+1 et Force, entre elles ainsi qu'avec des ordonnancements aléatoires (réalisés sur 100 tests).*

	MCF		*Band-Width*		*MCS*		*MCS+1*		*Force*		*aléatoire (100 tests)*		
Instance	nœuds	tps	nœuds	tps	nœuds	tps	nœuds	tps	nœuds	tps	nœ. min.	nœ. moy.	%réus.
VCSP↦SLDD$_+$													
Sml. Price Only	105	0,3s	40	0,2s	36	0,2s	263	0,6s	351	0,5s	31	1 547	100%
Med. Price Only	777	0,6s	312	0,5s	169	0,4s	20 091	1 078s	3 362	11,4s	527	7 872	68%
Big Price Only	m-o	-	46 415	25s	3 317	3,8s	-	t-o	-	t-o	-	-	0%
Sml. Hard Only	3 632	1,3s	4 418	0,9s	2 693	0,9s	2 158	0,9s	2 486	1,0s	3 431	7 205	100%
Med. Hard Only	8 914	1,4s	10 607	1,3s	5 971	1,3s	2 477	1,4s	4 038	1,6s	5 895	16 911	100%
Big Hard Only	m-o	-	286 713	68s	169 355	42s	211 848	71s	m-o	-	-	-	0%
Small	3 100	1,2s	4 349	1,0s	2 344	1,0s	1 744	0,8s	3 415	1,2s	2 912	7 698	100%
Medium	5 660	1,5s	11 700	1,6s	6 242	1,4s	3 238	1,6s	13 603	1,5s	7 324	17 435	100%
Big	m-o	-	326 884	112s	196 098	71s	73 702	35s	m-o	-	-	-	0%
Bayes↦SLDD$_\times$													
Asia	35	0,06s	29	0,06s	23	0,06s	29	0,06s	25	0,06s	26	58	100%
Car-starts	60	0,1s	40	0,09s	40	0,09s	47	0,09s	41	0,09s	97	211	100%
Alarm	m-o	-	5 843	0,8s	1 301	0,5s	3 198	0,5s	7 054	1,0s	-	-	0%
Hailfinder25	m-o	-	m-o	-	15 333	1,3s	-	t-o	139 172	114s	-	-	0%

Sur les cinq heuristiques d'ordonnancement, deux sortent du lot.

Tout d'abord il apparaît que **MCS** est la seule heuristique qui permette de compiler chacune des instances testées. Toujours parmi les meilleures, tant du point de vue de la taille du diagramme généré que de celui du temps de calcul, elle est la plus régulière sur ces instances.

L'heuristique **MCS+1** est, quant à elle, très irrégulière. Si c'est généralement elle qui donne les meilleurs résultats sur les problème composés partiellement ou exclusivement de contraintes dures (problèmes *hard Only* et problèmes « complets »), il est à noter d'importantes contre-performances quand il s'agit de problèmes constitués majoritairement ou exclusivement de contraintes valuées (problèmes *Price Only* et réseaux bayésiens).

Les heuristiques **MCF**, **Band-width** et **Force** sont moins intéressantes. L'heuristique **MCF** est une heuristique plutôt orientée compilation de contraintes dures, avec parfois de bon résultats lorsqu'il s'agit de problèmes « complets ». L'heuristique **Band-width** donne des résultats moyens sur l'ensemble des instances et **Force** enchaîne les contre-performances sur la majorité des instances.

Au vu de ces résultats, nous n'utiliserons que les heuristiques **MCS** et **MCS+1** dans la suite des expérimentations. Nous préconisons également d'utiliser en premier lieu l'heuristique **MCS** lors de la compilation de nouveaux problèmes en raison de sa régularité.

Heuristiques d'ordonnancement de contraintes

L'ordonnancement des contraintes a une influence directe sur le temps de compilation mais aussi sur la capacité de compilation d'un problème. En effet, suivant l'ordonnancement utilisé, la taille maximale atteinte par le diagramme peut provoquer un dépassement de mémoire, ou bien le temps de compilation peut dépasser la valeur de « *time-out* » fixée.

Les instances sont ici compilées avec leur heuristique d'ordonnancement de variables donnant le meilleur résultat, i.e. le diagramme le plus succinct, et l'on mesure le temps de compilation pour chacune des trois heuristiques présentées section à la 5.3. Ces heuristiques seront également comparées à « l'ordre naturel » déjà utilisé pour la comparaison des heuristiques d'ordonnancement de variables, ainsi qu'à un ordonnancement aléatoire des contraintes réalisé sur 100 tests, dont ont été extraits le temps moyen de compilation, le meilleur temps, et le pourcentage de réussite.

Les résultats sont présentés en table 6.2.

Table 6.2 – *Comparaison de l'ordre naturel et des heuristiques BCF, de tri par « dureté », de tri par « difficulté » de compilation, ainsi qu'avec des ordonnancements aléatoires (réalisés sur 100 tests).*

	heur. ordo.	nœuds	ordre nat.	BCF	difficulté	dureté	*aléatoire (100 tests)*		
Instance	variables		tps	tps	tps	tps	tps min	tps moy	% réus.
VCSP↦SLDD$_+$									
Sml. Price Only	MCS	36	0,2s	0,2s	0,2s	0,2s	0,2s	0,2s	100%
Med. Price Only	MCS	169	0,4s	0,4s	0,4s	0,4s	0,3s	0,4s	100%
Big Price Only	MCS	3 317	4,6s	4,9s	3,6s	5,9s	3,1s	3,4s	100%
Sml. Hard Only	MCS+1	2 158	1,2s	1,2s	1,4s	1,2s	0,7s	1,1s	100%
Med. Hard Only	MCS+1	2 477	1,5s	3,9s	1,5s	1,7s	1,2s	1,7s	100%
Big Hard Only	MCS	169 335	43s	m-o	35s	m-o	-	-	0%
Small	MCS+1	1 744	0,9s	1,2s	0,9s	0,9s	0,9s	1,3s	100%
Medium	MCS+1	3 238	1,4s	4,0s	1,3s	1,3s	1,1s	20,7s	100%
Big	MCS+1	73 702	33s	m-o	41s	47s	101s	259s	8%
Bayes↦SLDD$_\times$									
Asia	MCS	23	0,07s	0,07s	0,08s	0,07s	0,06s	0,06s	100%
Car-starts	MCS	40	0,11s	0,10s	0,10s	0,10s	0,08s	0,08s	100%
Alarm	MCS	1 301	0,6s	0,6s	0,7s	0,5s	0,3s	0,4s	100%
Hailfinder25	MCS	15 333	1,9s	4,5s	4,5s	1,8s	1,2s	1,4s	100%
bonus									
Med. Price Only	MCS+1	20 091	1 114s	t-o	t-o	t-o	100s	706s	90%

Ici, la comparaison des heuristiques d'ordonnancement de contraintes avec les ordonnancements aléatoires n'est pas excellente. En effet, on retrouve souvent, au statut de plus rapide, une des compilations effectuée avec un ordonnancement aléatoire.

Pour les instances de type réseau bayésien, contraintes valuées seules ou contraintes dures seules, l'ordonnancement aléatoire est même meilleur en moyenne que l'ensemble des heuristiques [2].

Ces heuristiques semblent cependant bonnes pour les problèmes « complets » (**Small**, **Medium** et **Big**), ce qui reste l'objectif principal.

L'utilisation de l'ordre naturel des contraintes donne de bons résultats autant dans les instances de réseaux bayésiens qu'avec les VCSP. Si ces contraintes n'ont pas été ordonnées aléatoirement dans les fichiers d'instances, alors, il peut être intéressant de tester cet ordonnancement.

Cependant, les disparités au niveau des résultats montrent l'importance d'une bonne heuristique d'ordonnancement des contraintes. Si les heuristiques basées sur la « difficulté » de compilation des contraintes ou la dureté des contraintes semblent convenir aux instances **Small**, **Medium** et **Big**, elles ne sont pas toujours satisfaisantes. L'ordonnancement de contraintes d'une instance quelconque reste un problème ouvert et il semble que chaque instance ou famille d'instances nécessite une heuristique différente appropriée.

Même si cela n'a aucune influence sur le diagramme après compilation, et donc sur la suite des expérimentations, nous utiliserons toujours par la suite l'heuristique donnant le plus faible temps de compilation pour chaque instance.

6.1.3 Efficacité spatiale des VDD

Nous avons voulu comparer la taille des diagrammes obtenus pour les différents langages considérés. Les jeux d'essai ont été compilés sous la forme de SLDD (SLDD_+ pour les VCSP et $\text{SLDD}_\times$ pour les réseaux bayésiens) puis traduits dans les autres langages. Ils ont été compilés avec leurs meilleures heuristiques, à savoir **MCS** pour les réseaux bayésiens et les problèmes type « Price Only » et **MCS+1** pour les problèmes « *classiques* ».

2. Notez en bonus dans le tableau 6.2 la compilation de l'instance **Medium Price Only** avec l'heuristique d'ordonnancement de variables MCS+1 dont aucune des heuristiques d'ordonnancement des contraintes ne permet la compilation, alors que l'ordonnancement aléatoire réussit dans 90% des cas.

Table 6.3 – *Compilation de problèmes de configuration en $SLDD_+$ et transformations en ADD, $SLDD_\times$ et AADD*

VCSP	$SLDD_+$		ADD	$SLDD_\times$	AADD
Instance	nœuds	temps	nœuds	nœuds	nœuds
Small Price only	36	0,2s	4 364	3 291	36
Medium Price only	169	0,4s	37 807	33 587	168
Big Price only	3 317	3,6s	m-o	m-o	3 317
Small	1 744	0,9s	28 971	19 930	1 744
Medium	3 238	1,3s	463 383	354 122	3 156
Big	73 702	34s	m-o	m-o	73 702

Table 6.4 – *Compilation de réseaux bayésiens en $SLDD_\times$ et transformations en ADD, $SLDD_+$ et AADD*

Rés. bay.	$SLDD_\times$		ADD	$SLDD_+$	AADD
Instance	nœuds	temps	nœuds	nœuds	nœuds
Asia	23	$0,07s$	415	216	23
Car-starts	40	$0,1s$	42 741	19 632	40
Alarm	1 301	$0,5s$	m-o	m-o	1 301
Hailfinder25	15 333	1,8s	m-o	m-o	15 331

Les tableaux 6.3 et 6.4 indiquent les tailles obtenues pour la représentation des problèmes de configuration et des réseaux bayésiens sous la forme de ADD, $SLDD_+$, $SLDD_\times$ et AADD, ainsi que, à titre indicatif, les temps de compilation. Le temps consommé par les transformations n'est pas une information importante ici car il n'est aucunement lié ou comparable aux temps de compilation initiaux, et il ne dépend que de la taille maximale atteinte au cours de la transformation.

Ces expérimentations confirment en pratique les résultats théoriques de compacité des différents langages obtenus dans les chapitres précédents. 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'instance 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, de gain pratique en termes de compacité. Autrement dit, l'utilisation d'un AADD par rapport à un SLDD_{+} (resp. $\text{SLDD}_{\times}$) n'offre pas de réelle amélioration pour la compilation des problèmes de configuration avec coût (resp. de réseaux bayésiens) [3].

Utilité des ADD

Nous pouvons observer à travers les expérimentations que le langage ADD atteint rapidement ses limites. Avec des tailles de diagramme de très loin supérieures aux SLDD et AADD, beaucoup de requêtes utiles à la recommandation étant de complexité linéaire dans la taille de la représentation, le langage ADD n'est pas adapté à nos besoins. Il sera donc abandonné durant les expérimentations à venir.

6.2 Configuration de produits

Ce que l'utilisateur doit pouvoir faire :

- sélectionner les variables dans l'ordre de son choix
- choisir une alternative sur une variable (ou bien éliminer une ou plusieurs alternatives)
- revenir sur l'un de ses choix

Le programme doit :

- ne lui présenter que les alternatives pouvant conduire à une solution
- lui donner à tout moment la valuation optimale possible compte tenu de la configuration en cours
- pouvoir lui donner la valuation optimale associée à chacune des alternatives possibles d'une variable

3. Lors de la compilation de réseaux bayésiens en ADD et AADD, nous obtenons des représentations nettement moins succinctes que celles décrites dans [Sanner et 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.

6.2.1 Procédure

Il est important, tout au long d'une configuration, de mettre à jour les domaines courants des variables, afin de s'assurer que le problème reste Globalement Inversement Consistant (GIC), c'est-à-dire que l'ensemble des valeurs du domaine de l'ensemble des variables peuvent conduire à une solution (voir définition en annexe, section A).

Deux méthodes sont possibles pour mettre pour mettre en œuvre ce processus. La première consiste à modifier la structure de représentation après chaque choix de l'utilisateur. Les arcs correspondants aux alternatives éliminées par l'utilisateur sont supprimés, et le diagramme est normalisé, supprimant ainsi les pans du diagramme inaccessibles. La deuxième consiste à déclarer seulement les arcs éliminés comme inaccessibles. La structure de représentation n'est pas modifiée, et aucune procédure de normalisation n'est à effectuer. Une procédure de propagation, similaire à la procédure de normalisation, permet de mettre à jour les informations sans pour autant modifier ni la structure, ni les valuations portées par les arcs.

La première méthode a pour avantage de réduire la taille de la représentation au fur et à mesure de la configuration. Cependant, aucun retour en arrière n'est possible. Il est alors nécessaire de réinitialiser la configuration en rechargeant la structure de représentation initiale. La deuxième méthode a pour avantage de ne pas modifier la structure, le retour arrière est possible et la procédure de propagation est moins lourde que la procédure de normalisation car la structure n'est pas modifiée. De plus, la réinitialisation d'une configuration ne nécessite pas de recharger la structure initiale, mais seulement d'effacer les informations ajoutées aux arcs.

La deuxième méthode est pour nous plus intéressante car elle est plus rapide en début de configuration (là où les traitements sont les plus longs), elle permet un retour en arrière sur les conditionnements (l'utilisateur peut revenir sur ses choix) et elle permet de recharger le problème de façon plus rapide. C'est donc celle que nous suivrons.

Propagation

La procédure de propagation est l'opération de base de la configuration de produit. Elle est lancée après chaque conditionnement et calcule la valuation minimale, maximale et le domaine courant de chaque variable (et donc s'assurer de la GIC).

Cette procédure remplace la procédure de normalisation, avec pour avantage d'être plus légère (car la structure n'est pas modifiée) et non destructrice.

Elle est réalisé comme suit. On associe à chaque nœud du VDD considéré ($SLDD_+$, $SLDD_\times$ ou AADD) deux valeurs, *min* et *max*. On attribue arbitrairement à *min* le titre de « valeur descendante » et à *max* le titre de « valeur ascendante ». Comme le diagramme n'est pas nécessairement normalisé, pour qu'un nœud soit accessible, il faut vérifier qu'il existe un chemin de la racine jusqu'à ce nœud et un chemin de ce nœud jusqu'au nœud terminal.

La valeur *min* associée à un nœud correspond alors au chemin de valuation minimale (plus l'offset) qui existe entre le nœud racine et ce nœud, ou la non valeur *null* si aucun chemin n'existe de la racine à ce nœud. La valeur *max* associée à un nœud correspond au chemin de valuation maximale qui existe entre le nœud terminal et ce nœud, ou *null* s'il n'y a aucun chemin du nœud terminal à ce nœud. La valuation minimale possible du diagramme est donc égale à la valeur *min* associée au nœud terminal, et la valuation maximale possible du diagramme est donc égale à la valeur *max* associée au nœud racine, plus l'offset.

Un nœud est considéré accessible si sa valeur *min* et sa valeur *max* sont différents de *null*. Le domaine courant d'une variable x est l'ensemble des valeurs d_i du domaine initial tel qu'il existe au moins un arc a_i de valeur d_i issu d'un nœud étiqueté x, et tel que ni l'arc a_i, ni les nœuds $In(a_i)$ et $Out(a_i)$ ne sont inaccessibles.

La première propagation (aussi appelée initialisation) consiste donc à parcourir (en largeur d'abord) l'ensemble du graphe dans l'ordre topologique, puis dans l'ordre topologique inverse, afin d'initialiser ces deux valeurs sur l'ensemble du graphe.

Après un conditionnement ou une restauration (l'annulation du conditionnement d'une variable), on parcourt le graphe à partir des nœuds étiquetés par la variable concernée, dans l'ordre topologique pour mettre à jour la valeur *min*, et dans l'ordre topologique inverse pour mettre à jour la valeur *max*, jusqu'à ce que la propagation n'entraine aucune modification, ou jusqu'à atteindre le nœud racine ou le nœud terminal.

Notez que sur un AADD, la procédure doit être légèrement modifiée car si le diagramme n'est pas normalisé, la structure de valuation fait que les valuations minimale et maximale ne peuvent se calculer que du nœud terminal à la racine. La valeur *min* associée à chaque nœud est calculée du nœud terminal à la racine, comme la valeur *max*, mais on effectue quand même un parcours de la racine au nœud terminal pour vérifier « l'accessibilité descendante ». Cette différence de méthode n'entraîne pas de différence de performance.

Table 6.5 – *Comparaison des temps d'une propagation initiale et complète sur des SLDD ou sur des AADD équivalents*

VCSP	$SLDD_+$	AADD	rapport
Sml. Price Only	11,8μs	13,5μs	1,14
Med. Price Only	63,9μs	84,5μs	1,32
Big Price Only	1,36ms	2,09ms	1,54
Small	222μs	281μs	1,27
Medium	487μs	578μs	1,19
Big	22,1ms	39,9ms	1,81
Bayes	$SLDD_\times$	AADD	rapport
Asia	29,0μs	32,3μs	1,11
Car-starts	61,5μs	75,6μs	1,23
Alarm	259μs	292μs	1,13
Hailfinder25	7,68ms	9,16ms	1,19

Intérêt des AADD

Si la section 6.1.3 nous a permis d'écarter les ADD trop gourmands en espace, aucune différence de taille n'est à noter sur nos jeux d'essai entre la compilation vers les langages cibles AADD et SLDD. Nous avons donc testé l'efficacité de ces deux langages dans un contexte de configuration de produit.

Nous calculons ici le temps de calcul associé à chacun de ces deux langages pour réaliser une propagation, l'opération élémentaire. Ici, nous comparons les temps de calcul de la propagation initiale.

Les résultats sont présentés en table 6.5.

Ces résultats nous montrent que, même si les représentations d'un problème dans les langages SLDD et AADD sont de même tailles, l'exploitation des SLDD est plus rapide. En effet, la structure de valuation du langage SLDD est plus légère que la structure de valuation du langage AADD, ce qui permet aux $SLDD_+$ d'être entre 1,2 et 1,8 fois plus rapides que les AADD, et aux $SLDD_\times$ d'être jusqu'à 1,2 fois plus rapides. Ce rapport de vitesse entre SLDD et AADD s'observe sur toute propagation, initiale ou non.

Le langage AADD n'apportant concrètement ici rien de plus que le langage SLDD, et étant donné les différences de rapidité entre les deux langages, nous ne nous concentrerons par la suite que sur les représentations sous la forme SLDD.

6.2.2 Protocole

Un protocole de validation a été mis en place dans le cadre du projet BR4CP, visant à tester et comparer les différents solveurs réalisés au cours de ce projet [4]. Ce protocole de validation reprend le comportement que pourrait avoir un utilisateur en ligne souhaitant configurer un produit. Il se décline en trois variantes.

1ère variante : greedy configuration (GC)

Il s'agit d'un cas simple dans lequel l'utilisateur fait ses choix dans le domaine courant des variables (c'est aux solveurs qu'il incombe de ne proposer à l'utilisateur que le domaine courant de chacune des variables). Il n'y a ici aucun retour en arrière et l'on va gloutonnement à une solution.

Cette variante se découpe en deux phases :

A. Phase d'initialisation :
- Obtenir les domaines initiaux en assurant la cohérence inverse globale
- Calculer les valuations minimale et maximale du produit fini.

B. Phase d'affectation :
- Choisir aléatoirement une variable v non affectée (de domaine courant > 1)
- Choisir aléatoirement une valeur de v appartenant à son domaine courant
  - Affecter cette valeur
  - Obtenir les domaines courants en assurant la cohérence inverse globale
  - Calculer les valuations minimale et maximale du produit fini compte tenu de la configuration courante
- Répéter la phase B. en boucle tant qu'il reste des variables non affectées

Cette variante peut être scénarisée : lors de la phase B, le choix de la variable v et de sa valeur suivent un scénario préalablement établis. Le suivi de ce scénario permet de vérifier l'exactitude des domaines courants ainsi que des valuations minimales et maximales donnés par les solveurs, ces données étant connu à l'avance par un programme tiers de vérification.

Cette variante se décline en deux versions : la version valuée présentée ci-dessus (GC-P), et la version non valuée (GC-U), ne requérant pas le calcul des valuations minimale et maximale.

4. Toutes les informations ainsi que les scénarios peuvent être trouvés sur la page http://www.irit.fr/~Helene.Fargier/BR4CP/Protocole.html

2ème variante : greedy configuration at min price (GCM)

L'utilisateur choisit ici toujours l'alternative conduisant à une solution de valuation minimale. Il n'y a ici aucun retour en arrière, on va donc glouton-nement à une solution optimale.

Cette variante se découpe de la même façon que la variante précédente, seul le choix de la valeur de v (phase B.2) change par rapport à la variante précédente :

A. Phase d'initialisation :
- Obtenir les domaines initiaux en assurant la cohérence inverse globale
- Calculer les valuations minimale et maximale du produit fini.

B. Phase d'affectation :
- Choisir aléatoirement une variable v non affectée (de domaine courant > 1)
- Choisir aléatoirement une valeur de v parmi les valeurs pouvant conduire à une solution de prix minimale
  - Affecter cette valeur
  - Obtenir les domaines courants en assurant la cohérence inverse globale
  - Calculer les valuations minimale et maximale du produit fini compte tenu de la configuration courante
- Répéter la phase B. en boucle tant qu'il reste des variables non affectées

Cette variante peut également être scénarisée : lors de la phase B, le choix de la variable v et de sa valeur suivent un scénario préalablement établis. Le suivi de ce scénario permet de vérifier l'exactitude des domaines courants et des valuations minimales et maximales données par les solveurs, ainsi que des valeurs de v conduisant à une solution optimale proposé par le solveur.

3ème variante : full configuration protocol (FCP)

Cette variante se veut plus complète que les variantes précédentes. L'utilisateur simulé ne fait plus ses choix dans le domaine courant des variables, mais dans le domaine initial des variables. Si cette valeur choisie n'appartient pas au domaine courant, alors l'utilisateur revient sur ses choix précédents jusqu'à réapparition de cette valeur.

Cette variante se découpe en trois phases :

A. Phase d'initialisation :
- Obtenir les domaines initiaux en assurant la cohérence inverse globale
- Calculer les valuations minimale et maximale du produit fini.

B. Phase d'affectation :
- Choisir aléatoirement une variable v non affectée (de domaine courant > 1)
- Choisir aléatoirement une valeur de v appartenant à son domaine initial
- Si cette valeur n'appartient pas au domaine courant de v, aller à C.

Sinon
- Affecter cette valeur
- Obtenir les domaines courants en assurant la cohérence inverse globale
- Calculer les valuations minimale et maximale du produit fini compte tenu de la configuration courante

- Répéter la phase B. en boucle tant qu'il reste des variables non affectées

C. Phase de restauration :
- Choisir aléatoirement une variable v déjà affectée
- La désaffecter
- Obtenir les domaines courants en s'assurant de la cohérence inverse globale
- Calculer les valuations minimale et maximale du produit fini compte tenu de la configuration courante

- Si la valeur à l'origine du conflit est réapparue, s'arrêter, sinon répéter la phase C.

Cette variante peut également être scénarisée : lors des phases B et C, le choix de la variable v et éventuellement de sa valeur, suivent un scénario préalablement établis. Le suivi de ce scénario permet de vérifier l'exactitude des domaines courants ainsi que des valuations minimales et maximales donnés par les solveurs.

Cette variante se décline en deux versions : la version valuée présentée ci-dessus (FCP-P), et la version non valuée (FCP-U), ne requérant pas le calcul des valuations minimale et maximale.

Interface Configurator.java

Afin de tester les solveurs en toute neutralité, le protocole de validation à été implémenté, intégrant une interface Java englobant l'ensemble des fonctions nécessaires à la configuration de produits, fonctions que chaque solveur, dont le notre, doit assurer. Voici les fonctions principales à implémenter :

```
void readProblem(String problemName)
void initialize()
void assignAndPropagate(String var, String val)
void unassignAndRestore(String var)

int minCost(), int maxCost()
Map<> minCostConfiguration(), Map<> maxCostConfiguration()
Map<> minCosts(String var), Map<> maxCosts(String var)

Set<> getCurrentDomainOf(String var)
Set<> getFreeVariables()
```

Les fonctions `readProblem()` et `initialize()` permettent respectivement de charger un problème et de réaliser l'initialisation. Ici, le problème peut être chargé de trois façons différentes. Il peut être compilé, il peut être chargé à partir d'un fichier de sauvegarde si la compilation de ce problème a déjà été effectuée, ou il peut être réinitialisé s'il a déjà été chargé durant cette session. L'initialisation consiste, elle, en une première propagation.

Les fonctions `assignAndPropagate()` et `unassignAndRestore()` permettent le conditionnement ou la restauration d'une variable. Le conditionnement consiste simplement à annoter comme tel les arcs rendus inaccessibles ou accessibles à nouveau. Une propagation doit être effectuée sur tout nœud/arc pouvant être impacté par ce changement.

`minCost()` et `maxCost()` renvoient respectivement la valuation minimale et maximale possible d'un produit fini compte tenu de la configuration actuelle, et les fonctions `minCostConfiguration()` et `maxCostConfiguration()` renvoient une de ces configurations correspondant à ces produits finis optimaux. La propagation ayant mis les informations à jour, il n'y a aucun calcul derrière ces fonctions, seulement l'affichage des résultats.

`minCostConfiguration()` et `maxCostConfiguration()` sont des fonctions qui associent à chaque valeur d'un domaine d'une variable la valuation minimale/maximale possible. Elles nécessitent quant à elles plus de calculs pour calculer les valuations de chacune des valeurs du domaine de notre variable.

Enfin les fonctions `getCurrentDomainOf()` et `getFreeVariables()` donnent respectivement la liste des valeurs appartenant au domaine courant d'une variable et la liste des variables dont le domaine courant et de taille supérieure à un. Encore une fois, la procédure de propagation a déjà fait tout le boulot.

Programme de test

Le programme de test, réalisant le protocole et unifiant tout solveur ayant implémenté l'interface a été réalisé par une tierce personne [5]. Ce programme exécute les différentes variantes, en suivant un scénario ou via des tirages aléatoires, sur le solveur demandé. Il vérifie, dans les versions scénarisées, la véracité des réponses des solveurs et, dans les versions aléatoires, la concordance entre les différents solveurs testés. De plus, sont mesurés les temps d'exécution des phases A (initialisation), B (affectations) et éventuellement C (restaurations).

6.2.3 Résultats

L'ensemble des tests a été effectué (versions scénarisés et aléatoires, versions valués et non valués), cependant nous ne présenterons ici que les résultats des expérimentations scénarisées et valuées, les autres cas donnant des résultats similaires.

Nous avons lancé les expérimentations ne concernant que notre compilateur sur la même machine que celle utilisée pour la réalisation des expérimentations présentées section 6.1.

Chaque scénario est composé de 1 000 simulations de configurations, et ces scénarios portent sur les instances **Small**, **Medium** et **Big**, dans leurs version valuée.

Nous présentons et discutons dans un premier temps les résultats de notre solveur (le compilateur SALADD), puis nous le comparons brièvement aux autres solveurs développés durant ce projet. Les autres solveurs utilisant des approches non compilées, nous attendons des résultats meilleurs que ceux offerts par les autres solveurs (sinon c'est que nous avons fait tout ce travail de compilation pour rien).

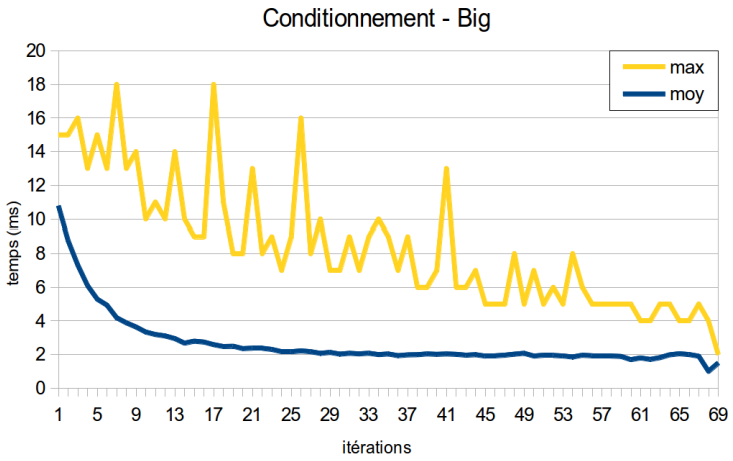
Initialisation et conditionnement

La variante GC-P est constituée de deux phases, la phase A d'initialisation (toujours identique sur les 1 000 simulations), et d'une phase B d'affectations, constituée d'autant de conditionnements que nécessaire à ce que plus aucune variable ne soit libre de choix.

Cette variante nous permet donc de calculer le temps d'initialisation (qui correspond à une propagation complète), le temps moyen de la phase d'affectation, mais surtout le temps moyen et le temps maximal d'un conditionnement,

5. Un stagiaire !

Figure 6.1 – *Temps moyen et maximal (en ms) de conditionnement du problème **Big** en fonction du nombre d'itération sur la variante GC-P par notre compilateur.*



ce qui correspond au temps que va devoir attendre un utilisateur configurant son produit en ligne pour que le problème s'actualise après chaque choix.

Voici les résultats obtenus par notre approche :

Instance **Small** :
-Phase A : 0,22ms (temps invariant)
-Phase B : 0,77ms pour 12,08 conditionnements (moyenne)
-Conditionnement : 0,065ms (moyenne) 4,3ms (maximum)

Instance **Medium** :
-Phase A : 0,49ms (temps invariant)
-Phase B : 1,40ms pour 10,93 conditionnements (moyenne)
-Conditionnement : 0,12ms (moyenne) 7,8ms (maximum)

Instance **Big** :
-Phase A : 22,1ms (temps invariant)
-Phase B : 147ms pour 49,42 conditionnements (moyenne)
-Conditionnement : 2,98ms (moyenne) 18ms (maximum)

La figure 6.1 nous montre l'évolution du temps moyen et maximal de conditionnement pour le problème **Big** selon l'avancement de la configuration.

Notez que le temps de conditionnement (figure 6.1) diminue au fur et à mesure des itérations. cette caractéristique se retrouve également sur les instances **Small** et **Medium**. Cela est dû au fait que le nombre de nœuds an-

notés « inaccessibles », à cause des conditionnements précédents, augmente. Le nombre de nœuds à traiter est donc de plus en plus faible au fur et à mesure de l'avancement de la configuration.

Avec des temps de calcul qui n'excèdent jamais 18ms (22ms en comptant la phase d'initialisation), nous pouvons conclure que cette méthode semble adaptée à la configuration en temps réel en ligne.

Restauration

La variante FCP-P intègre une phase de restauration (phase C) en plus des phases A et B identiques à la variante GC-P. Les temps d'initialisation et de conditionnement étant identiques, nous ne nous interesserons ici qu'à la phase de restauration.

Voici les résultats obtenus par notre approche :

Instance **Small** :
-Phase C : 0,73ms pour 3,15 restaurations (moyenne)
-Restauration : 0,19ms (moyenne) 3,82ms (maximum)

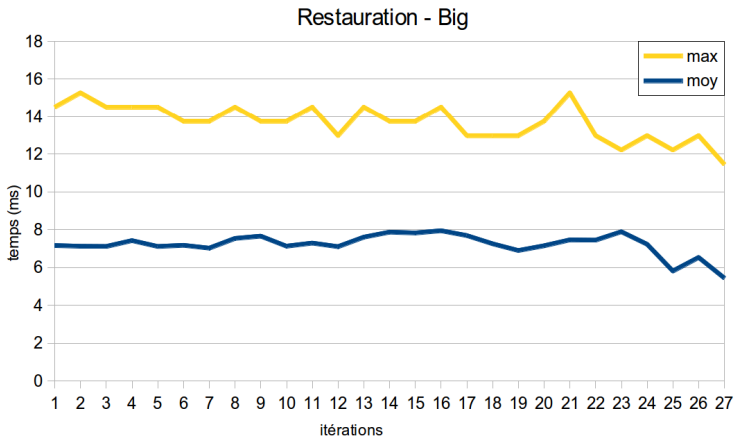
Instance **Medium** :
-Phase C : 0,83ms pour 2,78restaurations (moyenne)
-Restauration : 0,30ms (moyenne) 5,34ms (maximum)

Instance **Big** :
-Phase C : 52,6ms pour 7,24 restaurations (moyenne)
-Restauration : 7,26ms (moyenne) 15,3ms (maximum)

La figure 6.2 nous montre l'évolution du temps moyen et maximal de restauration du problème **Big**.

Ici, le temps de restauration (figure 6.1) ne diminue ni n'augmente au fur et à mesure des itérations. Cette caractéristique se retrouve également sur les instances **Small** et **Medium**. Cela est dû au fait que le les nœuds doivent être mis à jour qu'ils soient annotés « inaccessibles » ou pas, car ils peuvent redevenir accessibles, et si ils le sont déjà, les valuations maximale et minimale peuvent avoir besoin d'être mises à jour. Le nombre d'itérations déjà effectué n'a aucune influence.

Figure 6.2 – *Temps moyen et maximal (en ms) de restauration du problème **Big** en fonction du nombre d'itérations sur la variante FCP-P, par notre compilateur.*



min-marginalisation

La variante GCP intègre une phase de marginalisation par l'opérateur *min*. On veut déterminer avant chaque choix la valuation minimale possible associée à chacune des alternatives.

Cette opération est effectuée sur un SLDD_+ pour une variable v située en $i$ème position, en calculant, pour chacun des nœuds étiquetés par la variable v, le chemin de valuation minimale de la racine à ce nœud et, pour chacun des nœuds étiquetés par la variable w située en position $i+1$, le chemin de valuation minimale de ces nœuds au nœud terminal. Il ne reste ensuite qu'à calculer la combinaison minimale entre les nœuds étiquetés v et les nœuds étiquetés w pour chacune des alternatives de v.

Voici les résultats obtenus par notre approche :

Instance **Small** :
-Marginalisation : 0,23ms (moyenne) 2,63ms (maximum)

Instance **Medium** :
-Marginalisation : 0,49ms (moyenne) 4,55ms (maximum)

Instance **Big** :
-Marginalisation : 21,5ms (moyenne) 136ms (maximum)

On peut expliquer un temps de marginalisation assez long par l'opérateur *min* par la taille de domaine parfois importante de certaines variables (allant jusqu'à 324 pour l'instance **Big**). Cependant, nous restons dans des temps largement raisonnables pour de la configuration en ligne.

Chargement d'une instance

Le temps de lecture/chargement/rechargement d'un problème n'a pas été abordé lors de ce protocole de test mais peut avoir son importance.

Il existe trois moyens de « charger » un problème en mémoire. La première consiste à compiler le problème, mais cette méthode n'a besoin d'être faite qu'une seule fois et hors ligne, la 2ème consiste à lire un fichier de sauvegarde de la forme compilée, et la 3ème consiste à réinitialiser le problème déjà chargé, sous réserve que la structure n'ait pas été modifiée.

Voici les résultats obtenus par notre approche :

Instance **Small** :
-Lecture d'un fichier de sauvegarde : 5,9ms
-Réinitialisation : 0,63ms

Instance **Medium** :
-Lecture d'un fichier de sauvegarde : 10,9ms
-Réinitialisation : 1,17ms

Instance **Big** :
-Lecture d'un fichier de sauvegarde : 285ms
-Réinitialisation : 33,5ms

Récapitulatif

Instance **Small** :
-Compilation : 0,9s
-Chargement : 5,9ms (1er) / 0,63ms (autres)
-Première propagation : 0,22ms
-Conditionnement : 0,065ms (moyenne)
-Marginalisation : 0,23ms (moyenne)
-Restauration : 0,19ms (moyenne)

Instance **Medium** :
-Compilation : 1,3s
-Chargement : 10,9ms (1er) / 1,17ms (autres)
-Première propagation : 0,49ms
-Conditionnement : 0,12ms (moyenne)
-Marginalisation : 0,49ms (moyenne)
-Restauration : 0,30ms (moyenne)

Instance **Big** :
-Compilation : 33s
-Chargement : 285ms (1er) / 33,5ms (autres)
-Première propagation : 22,1ms
-Conditionnement : 2,98ms (moyenne)
-Marginalisation : 21,5ms (moyenne)
-Restauration : 7,26ms (moyenne)

Comparaison avec les autres solveurs

Les expérimentations qui vont suivre ont été effectuées sur un ordinateur moins puissant que précédement, il s'agit d'un processeur Intel Xeon cadencé à 1,6GHz (contre 2,7GHz précédemment). Nous voulons ici comparer notre approche compilée à deux autres solveurs qui suivent une approche non compilée. Ces solveurs sont SAT4J [Le Berre et Parrain, 2010], solveur SAT développé en Java et ABSCON [Lecoutre et Tabary, 2006, 2008], solveur CSP également developpé en Java.

La comparaison effectuée nous a également permis de vérifier que les trois solveurs donnent toujours les mêmes résultats, ce qui tend à prouver qu'ils sont tous corrects. Le solveur ABSCON ne prend pas en compte la valuation, cependant les solveurs SALADD et SAT4J donnant des résultats similaires avec et sans valuation, nous effectuerons cette comparaison sur les scénarios non valués.

La figure 6.3 donne le temps de conditionnement en fonctions du numéro d'itération sur le scénario GC-U et sur l'instance **Big** pour les trois solveurs. Sont affichés le temps moyen et le temps maximal pour chaque itération, ainsi que le temps moyen global de conditionnement.

Le tableau 6.6 donne les temps moyens et maximaux pour les instances **Small**, **Medium** et **Big** pour les trois solveurs sur les requêtes principales [6].

6. Malheureusement, les tests effectués par le programme de test générique sur la variante GCM sont erronés. Si nous avons pu refaire ces tests sur le solveur SALADD, nous devrons nous passer des résultats des autres solveurs sur cette variante du protocole.

Figure 6.3 – *Temps moyen et maximal (en ms) de conditionnement du problème **Big** en fonction du nombre d'itérations pour les trois solveurs SALADD, ABSCON et SAT4J, sur la variante GC-U.*

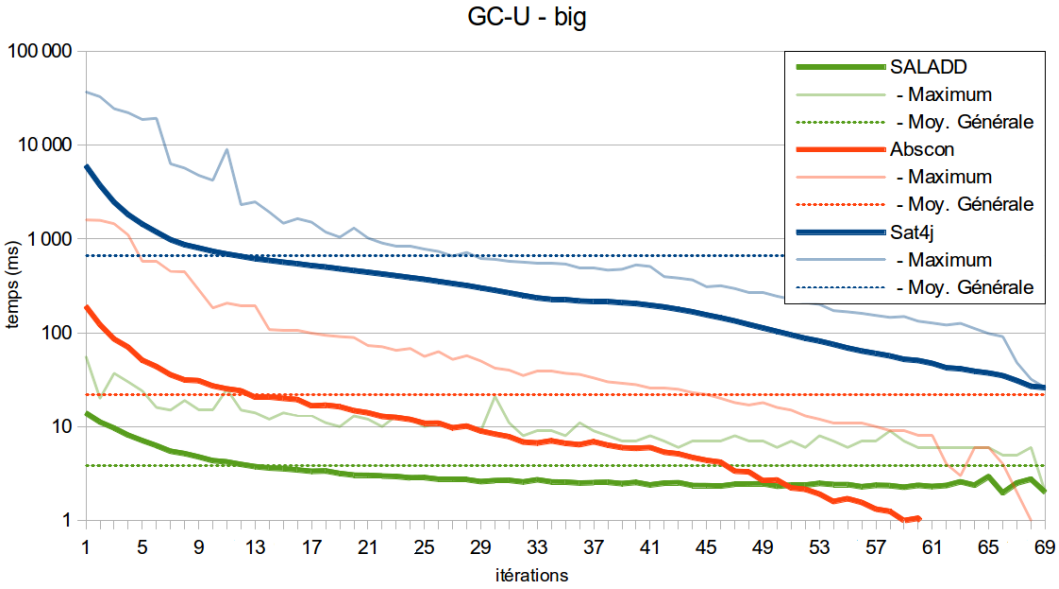


Table 6.6 – *Comparaison des temps d'initialisation (Phase A), de conditionnement et de restauration pour les solveurs SALADD, ABSCON et SAT4J*

Operation	SALADD tps (tps max)	ABSCON tps (tps max)	SAT4J tps (tps max)
Initialisation			
- Small	0,35ms	20,5ms	45,1ms
- Medium	0,75ms	35,7ms	105ms
- Big	32,3ms	1,53s	13,9s
Conditionnement			
- Small	0,2ms (7ms)	1,83ms (43ms)	13,9ms (144ms)
- Medium	0,3ms (10ms)	2ms (30ms)	26,7ms (236ms)
- Big	4ms (56ms)	22ms (1,6s)	663ms (36,8s)
Restauration			
- Small	0,25ms (4ms)	3,63ms (37ms)	-
- Medium	0,39ms (7ms)	5,2ms (67ms)	-
- Big	9,6ms (19ms)	264ms (6,4s)	-

Ces résultats tendent à confirmer l'intérêt de l'utilisation d'une forme compilée. En plus de garantir des temps moyens sur chaque requête nettement plus faibles que les autres méthodes auxquelles nous avons comparé notre solveur, la forme compilée garantit des complexités linéaires dans la taille de la forme compilée sur l'ensemble des requêtes, ce qui assure des temps au pire cas plus que raisonnables pour ce genre de problème.

Autre approche

Une autre approche, différente de ce protocole, est possible. En effet, afin de ne pas saturer la mémoire d'un configurateur dans le cas où beaucoup d'utilisateurs sont en train de configurer leurs produits en même temps, certains constructeurs tels que Renault préfèrent ne pas garder en mémoire la configuration courante, mais seulement l'historique des choix faits par l'utilisateur.

Lorsque l'utilisateur fait un nouveau choix (ou annule un de ses choix précédents), il faut alors réinitialiser la structure, la conditionner par l'ensemble des choix faits par l'utilisateur (historique et choix présent), et effectuer une propagation afin de calculer les informations intéressantes.

Ainsi, plusieurs personnes peuvent simultanément utiliser le programme sans en utiliser toute la mémoire.

Cette approche n'a pas été prise en compte par le protocole, et aucune variante ne ressemble à ce mode de fonctionnement. Nous pouvons cependant facilement calculer le temps de calcul pris par ce type d'approche en utilisant le solveur SALADD.

Les phases de rechargement/réinitialisation et de conditionnement sans propagation peuvent être faites en même temps, et sans augmentation significative de temps par rapport aux temps d'initialisation précédemment calculés. Le temps de la phase de propagation a été également calculé précédemment (première propagation) et l'intégralité des nœuds est toujours parcourue au cours de cette phase, donc aucune variation significative de temps par rapport au temps de phase A mesurés n'est à prévoir.

Le temps sera donc, dans le pire des cas, pour l'instance **Small** : $0,63 + 0,22 = 0,85ms$, pour l'instance **Medium** : $1,17 + 0,49 = 1,66ms$, et pour l'instance **Big** : $33,5 + 22,1 = 55,6ms$.

Chapitre 7

Autres applications

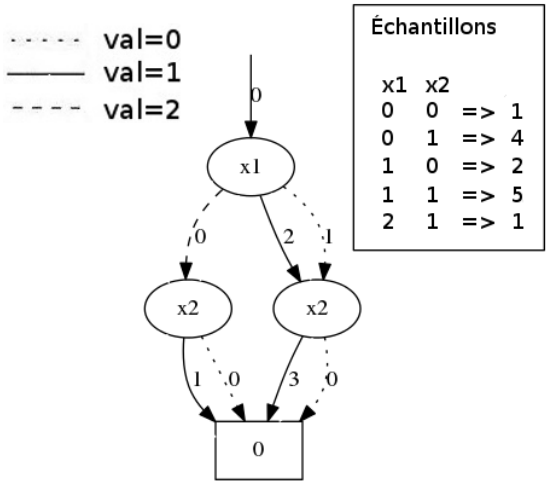
Au risque d'empiéter sur le chapitre des perspectives, nous allons dans ce chapitre aborder deux problèmes, différents de la configuration de produits, dans lesquels l'utilisation de SLDD et du compilateur SALADD peut se révéler intéressante. Le premier problème est un problème de recommandation pour lequel nous avons développé une méthode basée sur la compilation des historiques de vente, et l'exploitation de la forme compilée. Cette méthode qui en est au stade préliminaire sera comparée à une méthode plus classique utilisant des réseaux bayésiens. Le deuxième problème est l'utilisation de réseaux bayésiens compilés pour le calcul d'inférence (voir à la section 7.2).

7.1 Recommandation

En plus des jeux d'essais **Small**, **Medium** et **Big**, nous avions à notre disposition les historiques de vente, échelonnés sur une année, correspondants à chacune de ces instances. Ces historiques contiennent l'ensemble des choix faits par chacun des utilisateurs, que le produit ait été configuré par le configurateur de Renault, configuré par un conseiller de Renault, ou non configuré, c'est-à-dire acheté sur catalogue ou parmi les modèles déjà assemblés.

Ces historiques peuvent être vus comme un réseau de contraintes valuées, contenant une unique contrainte, dans laquelle chaque n-uplet (que nous appellerons « échantillons ») correspond à une configuration possible du produit, et la valuation associée correspond au nombre d'acheteurs pour ce modèle du produit (ou nombre de cet échantillon vendu). À ce titre, ces historiques peuvent être compilés dans le langage SLDD_+ de la même façon que précédemment (voir par exemple la figure 7.1).

Figure 7.1 – *Historique de vente portant sur deux variables, et compilé sous la forme d'un SLDD$_+$*



La contrainte portant toutes sur l'ensemble des variables (sinon cela signifierait que la valuation s'appliquerait à plusieurs chemins, et donc produits), aucune heuristique d'ordonnancement des variables ou des contraintes ne peut être appliquée (du moins, aucune des heuristiques présentées précédemment).

Le but est de pouvoir, tout au long de la configuration, conseiller l'utilisateur sur ses choix compte tenu des choix qu'il a déjà effectués et compte tenu des choix faits par les utilisateurs précédents (historiques de vente).

Basiquement, la méthode utilisée consiste à compiler l'historique de vente dans le langage SLDD$_+$, afin de disposer d'un SLDD représentant les contraintes (valuées et non valuées) s'appliquant au produit, et un SLDD représentant l'historique de vente. Lors de chaque choix effectué par l'utilisateur, on conditionne le SLDD représentant l'historique de vente de la même façon que l'on conditionne le SLDD représentant les contraintes. Lorsque l'utilisateur veut faire un nouveau choix sur une variable, on peut alors ajouter aux informations sur le domaine courant de la variable et les valuations optimales, une recommandation sur les valeurs de cette variable.

Cette recommandation est basée sur le principe de comptage d'échantillons et utilise plusieurs processus, dont la description va suivre.

7.1.1 Comptage pondéré

Dans notre cas, l'opération essentielle pour la recommandation consiste au comptage du nombre d'échantillons dans le diagramme.

Cas général

L'opération de comptage du nombre d'échantillons vendus ne se limite pas à un simple comptage de modèles (qui compte combien de modèles différents peuvent exister). Le calcul du nombre d'échantillons, que nous avons appelé « comptage pondéré » doit faire la somme du nombre de fois que chacun des modèles possibles de configuration a été vendu.

Cependant, le décompte de chacun des modèles possibles n'est pas nécessaire. La procédure suivante, associe deux valeurs *cpt* (représentant le comptage de modèles) et *px* (représentant le comptage pondéré) à chaque nœud. Cette méthode est de complexité linéaire dans la taille du SLDD_+.

Initialisation :

$$\begin{cases} cpt(root) = 1 \\ px(root) = \phi_0 \end{cases}$$

Pour chaque nœud :

$$\begin{cases} cpt(N) = \sum\limits_{\forall a | out(a) = N} cpt(In(a)) \\ px(N) = \sum\limits_{\forall a | out(a) = N} px(In(a)) + \phi(a) \times cpt(In(a)) \end{cases}$$

Résultat final :

$$\#\text{échantillons} = \textstyle\sum_{\vec{x}} f(\vec{x}) = px(leaf)$$

Voir un exemple de comptage pondéré à la figure 7.2.

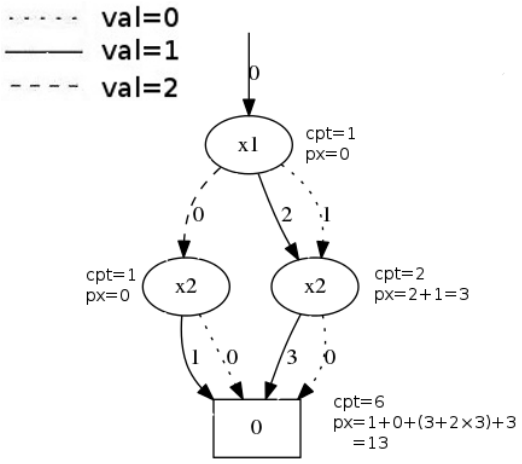
Comptage pondéré valeur par valeur

Afin de pouvoir recommander une valeur pour une variable, il convient de calculer, non pas le nombre d'échantillons global, mais la proportion dans laquel chacune des valeurs a été choisie.

En conditionnant successivement par chacune des valeurs d'une variable, et en comptant à chaque fois le nombre d'échantillons correspondants grâce à la fonction de comptage pondéré, on peut calculer, pour la variable considérée, la valeur la plus choisie et dans quelles proportions. Cette opération correspond à la requête de marginalisation par l'opérateur + (+**Marg**) sur la variable concernée.

Cette opération est cependant d'autant plus longue que la variable a de valeurs, et que cette variable est placée en début d'ordre dans l'ordonnancement des variables (les valeurs de *cpt* et *px* étant invariantes pour tout nœud situé en amont des nœuds étiquetés par la variable à analyser).

Figure 7.2 – *Calcul des valeurs cpt et px sur l'ensemble des nœuds d'un historique compilé. Le nombre d'échantillons total correspond à la valeur de px du nœud terminal, c'est-à-dire ici* 13.



7.1.2 Restauration de variables

Au fur et à mesure des choix de l'utilisateur, et des conditionnements effectués, le nombre d'échantillons diminue (on ne garde que les échantillons correspondant exactement aux choix faits par l'utilisateur). Tant que ce nombre d'échantillons reste élevé, il est possible de donner une recommandation fiable et basée uniquement sur les échantillons concordant avec la configuration courante.

Cependant, si le nombre d'echantillons devient faible voir nul (si la configuration actuelle est inédite), cette recommandation devient imprécise voire impossible.

Pour pallier ce problème, une approche consiste à ignorer certains choix précédents en restaurant des variables afin d'augmenter le nombre d'échantillons sur lesquels se baser.

Indépendance des variables

Quitte à ignorer certaines variables (choix précédents) lors de la recommandation, autant ignorer celles n'étant peu ou pas corrélées à la variable sur laquelle on souhaite faire une recommandation. Si par exemple le choix de la couleur de la carrosserie n'a aucune influence sur le choix de l'autoradio, alors ont peut ignorer la préférence émise sur la couleur le temps de recommander un modèle d'autoradio, afin d'augmenter le nombre d'échantillons sans pour autant perdre d'information déterminante.

On calcule alors l'indépendance de chaque couple de variables. Pour cela nous utilisons le test du χ^2 auquel on applique la correction de Yates [Yates, 1934]. On construit une table de contingence des occurrences pour tout couple de variables, cette table étant construite grâce à la fonction de comptage pondéré. On peut ensuite calculer un coefficient d'indépendance grâce au test du χ^2 pour chacun des couples.

Ces calculs pouvant être assez longs, les résultats sont sauvegardés pour pouvoir être rechargés plus rapidement.

Seuil et restauration

Un seuil a été défini. Lorsque le nombre d'échantillons tombe en dessous de ce seuil, il convient alors de restaurer des variables afin d'augmenter le nombre d'échantillons représentatifs. Actuellement, ce seuil a été fixé arbitrairement à 100 échantillons.

Les variables sont restaurées une à une, de manière gloutonne, de la plus indépendante à la moins indépendante, jusqu'à repasser au dessus du seuil. Une fois la recommandation faite, les variables restaurées sont conditionnées à nouveau suivant les choix de l'utilisateur.

7.1.3 Méthode de référence

Nous avons voulu comparer cette méthode à une méthode basée sur un apprentissage des historiques de vente en réseau bayésien. Cette méthode a été implémenté dans le cadre d'un stage. L'apprentissage des historiques de vente est effectué par le logiciel *R*, via le package *BNlearn* [Scutari, 2010].

L'apprentissage se déroule en deux phases. La première est une phase de construction de la structure du réseau bayésien par recherche de dépendance entre les variables. Cette construction est effectuée avec l'algorithme *min-max hill-climbing* [Tsamardinos *et al.*, 2006]. La deuxième phase consiste à définir l'orientation des arcs du réseau bayésien, et les probabilités associées aux variables.

Une fois les préférences des utilisateurs transformées en réseau bayésien, il ne nous reste plus qu'à réaliser un calcul d'inférence : étant donné les choix déjà effectués, on calcule la probabilité que l'utilisateur choisisse chacune des valeurs.

7.1.4 Protocole et résultats

Nous n'avons testé ces méthodes que sur l'instance **Small**. L'historique de vente est divisé en dix sous-ensembles, neufs sont utilisés pour l'apprentissage / la compilation, et le dernier pour la validation. Cette opération est effectuée dix fois afin de réaliser une validation croisée.

Pour chaque simulation (chaque échantillon) de l'ensemble de test, on définit aléatoirement un ordre sur les variables. On effectue une recommandation sur la variable suivante (ou la première variable si il s'agit de la première itération) suivant cet ordre aléatoirement établi. Si celle-ci coïncide avec la préférence de l'utilisateur, c'est un succès, sinon c'est un échec. Si à cause des contraintes, seulement une seule valeur est disponible, alors il n'y a pas besoin de recommandation, est cela ne compte ni pour un échec ni pour un succès. Dans tous les cas, le choix qui est finalement gardé, et par lequel on va conditionner, est la préférence initiale de l'utilisateur.

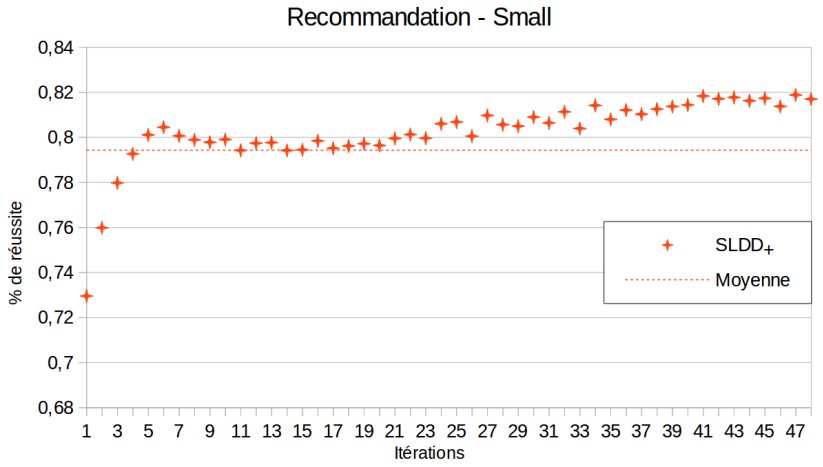
Les résultats offerts par les deux méthodes sont similaires. Notre méthode donne 79,43% de bonnes réponses, quand la méthode par apprentissage d'un réseau bayésien donne 79,46% de bonne réponses (en moyenne une simulation est composée de 13,5 recommandations). Cependant, notre méthode est plus rapide, avec en moyenne 140 millisecondes par simulation (soit environ 10,4ms par recommandation) contre 1,6 secondes par simulation pour la méthode par apprentissage d'un réseau bayésien (soit environ 126ms par recommandation).

La figure 7.3 donne le pourcentage de réussite pour les deux méthodes (les taux de réussite sont quasiment identiques) en fonction de l'avancement dans la configuration. Cette courbe montre bien l'intérêt de la recommandation, où plus nous possédons d'information sur les choix de l'utilisateur, plus la recommandation devient précise.

7.2 Compilation de réseaux bayésiens et inférence

L'inférence dans un réseau bayésien est l'opération qui consiste à calculer une probabilité, étant donné des informations observées. Cette opération est NP-difficile [Cooper, 1990]. Cependant après compilation d'un réseau bayésien sous la forme d'un $\text{SLDD}_{\times}$, cette requête devient de complexité linéaire dans la taille du $\text{SLDD}_{\times}$.

Figure 7.3 – *Évolution du pourcentage de réussite de la méthode basée sur la compilation d'historique (l'autre méthode donne des résultats quasi-identiques) en fonction du nombre de choix déjà effectués par l'utilisateur.*



On retrouve ici la même problématique que pour la compilation de VCSP pour la configuration de produits, le problème passe de NP-difficile à P après une phase de compilation. La compilation peut donc être intéressante quand beaucoup d'opérations doivent être réalisées sur un même problème.

La méthode de calcul d'inférence sur un $\text{SLDD}_\times$ s'approche de la méthode de comptage pondéré sur un SLDD_+. On associe une valeur inf à chaque nœuds, et le résultat correspond à la valeur inf associé au nœud terminal. La méthode est la suivante.

Initialisation :

$$inf(root) = \phi_0$$

Pour chaque nœud :

$$inf(N) = \sum_{\forall a | out(a)=N} inf(In(a)) \times \phi(a)$$

Résultat final :

$$\text{inférence} = inf(leaf)$$

Nous avons déjà utilisé cette méthode de calcul d'inférence pour la recommandation par apprentissage de réseau bayésien (le réseau bayésien est appris grâce au logiciel R puis compilé par le compilateur SALADD). Notre méthode de calcul exact d'inférence donne des résultats temporels comparables, voire meilleurs, que certaines méthodes de calcul approché d'inférence, et garantit en plus un temps de réponse borné, adapté aux applications en ligne.

Chapitre 8

Conclusion

Nous avons, dans cette thèse, développé nos recherches selon trois axes : l'étude théorique des diagrammes de décisions valués et plus particulièrement les trois langages ADD, SLDD et AADD ; la compilation de problèmes CSP valués et des réseaux bayésiens vers ces langages cibles et l'exploitation théorique qui peut en être faite ; et enfin le développement du compilateur SALADD et les expérimentations qu'il nous a permis de réaliser.

Cette conclusion résume les contributions que nous avons apportées à chacun de ces trois axes, mais aussi les perspectives que nous voyons à ces travaux.

Contributions

Extension du cadre SLDD

Dans le but de nous réapproprier les différents langages de la famille des VDD, nous avons remanié le cadre de définition du langage SLDD pour proposer les SLDD étendus (e-SLDD). Ce nouveau cadre, qui a fait l'objet d'une publication [Fargier *et al.*, 2013c], redéfinit la structure de valuation associée aux SLDD, et met en place une procédure de normalisation donnant à toute formule exprimée dans ce langage une forme canonique. Ce cadre nous a permis d'établir une structure de valuation qui montre que le langage AADD peut être vu comme un SLDD particulier, et ainsi d'associer à ces deux langages des règles communes.

Carte de compilation

L'étude théorique des langages de la famille des VDD nous a permis d'établir une hiérarchie de compacité théorique entre ces langages. Nous avons étendu au cas non booléen le travail initié par Darwiche et Marquis [2002] visant à établir une carte de compilation. Étant les premiers à élargir cette dernière à des langages de représentation non booléens, nous avons dû redéfinir un ensemble de requêtes et transformations qui nous semblait convenir à ces langages. Cette carte de compilation a fait l'objet des publications [Fargier *et al.*, 2014a] et [Fargier *et al.*, 2014b], reprenant les résultats de compacité obtenus, les requêtes et transformations définies, ainsi qu'un ensemble de preuves démontrant pour chacune des requêtes et transformations si elle est satisfaite ou pas par chacun des langages.

Algorithme de compilation

Nous proposons également un algorithme de compilation de problèmes de type réseau de contraintes vers le langage SLDD avec une approche ascendante. Cet algorithme, proche de l'algorithme *apply* proposé par Bryant [1986] permet d'incorporer directement une table de contraintes valuées à un SLDD. Cet algorithme a prétention à maintenir autant que possible la taille du diagramme la plus petite possible tout au long de la compilation (la taille importante pouvant être atteinte au cours de la compilation étant la faiblesse de la compilation ascendante), et à minimiser le nombre d'opérations effctuées (i.e. de nœuds traités) au cours du processus d'ajout d'une contrainte à un SLDD.

Nous avons également étudié plusieurs heuristiques d'ordonnancement des variables déjà existantes, et développé de nouvelles heuristiques d'ordonnancement de variables et de contraintes, dans le but de minimiser au maximum la taille de représentation des problèmes compilés, et de minimiser les temps de compilation.

L'algorithme de compilation, ainsi que les heuristiques (bien que les plus récentes soient une exclusivité de cette thèse) ont été présentées dans les articles [Fargier *et al.*, 2013b] et [Fargier *et al.*, 2014c].

Compilateur SALADD

Une importante contribution de cette thèse réside dans le compilateur SALADD que nous avons développé. Il permet de tester et valider en pratique l'ensemble des théories que nous avons étudiées et des méthodes que nous avons proposées. Notre compilateur implémente l'algorithme de compilation ainsi que les différentes heuristiques, les données peuvent être représentées

autant sous la forme d'un SLDD que d'un AADD, et il traite la plupart des requêtes et des transformations étudiées.

Ce programme est pleinement fonctionnel et mis à disposition à l'adresse <www.irit.fr/~Helene.Fargier/BR4CP/CompilateurSALADD.html> . Son utilisation se veut à la fois simple et complète de par la mise en œuvre de plusieurs modes de fonctionnement. Un mode de fonctionnement en ligne de commande permet de compiler directement et simplement des réseaux de contraintes valués ou des réseaux bayésiens. Une bibliothèque plus complète et en Java permet la compilation d'un problème et l'exploitation de celui-ci, via un certain nombre de fonctions adaptées à la configuration de produits et pour la recommandation. Enfin, pour les utilisateurs les plus avancés, il est possible d'implémenter une interface afin de créer et tester ses propres heuristiques d'ordonnancement de variables et de contraintes.

Expérimentations

Un important travail d'expérimentation nous a permis, dans un premier temps, de valider le bon fonctionnement ainsi que les bonnes performances du compilateur grâce au protocole de test réalisé dans le cadre du projet ANR BR4CP, et dans un deuxième temps, d'étayer nos résultats théoriques par des résultats expérimentaux. Ces résultats expérimentaux nous ont permis de comparer les différentes heuristiques proposées et les différents langages de représentation. Ces résultats nous ont également amené à écarter les ADD comme langage cible potentiel, et de montrer que seul l'opérateur $+$ (resp. $\times$) est utile lors de la compilation de problèmes purement additifs (resp. multiplicatifs). Ils nous conduisent à conseiller l'utilisation des SLDD_{+} pour la compilation de VCSP et des $\text{SLDD}_{\times}$ pour la compilation de réseaux bayésiens. Enfin les expérimentations nous ont permis de confirmer l'intérêt de notre approche (la compilation) pour les problèmes de configuration de produit, et de proposer plusieurs méthodes efficaces, offrant une garantie de temps de réponse, adaptées aux problèmes de configuration de produits.

Perspectives

Beaucoup de travail reste à effecuer, autant d'un point de vue théorique que du côté du développement logiciel. Nous avons principalement étudié les langages ADD, SLDD et AADD, mais il serait intéressant d'établir la carte de compilation d'autres langages proches tels que les Circuits Arithmétiques les AOMDD ou les SDD, qui sont théoriquement plus compacts que les SLDD. D'une même façon, nous n'avons travaillé qu'avec des diagrammes de décisions purement déterministes. Renoncer au déterminisme de nos structures

pourrait théoriquement donner des structures exponentiellement plus petites. Il serait intéressant d'étudier les conséquences réelles sur la taille des représentations suivant les langages utilisés (Circuits Arithmétiques, AOMDD, SDD, VDD non déterministes) ainsi que les requêtes et transformations toujours réalisables par ces langages. Nous étendrions ainsi les travaux présentés dans [Amilhastre *et al.*, 2014] sur les MDD non-déterministes.

Nous avons développé plusieurs heuristiques d'ordonnancement et montré leur utilité. Cependant, le choix de l'heuristique, notamment l'heuristique d'ordonnancement des variables a une influence très importante sur la taille de la représentation obtenue. Nous pensons que l'utilisation d'heuristiques plus abouties (la majorité de nos heuristiques utilisent des approches gloutonnes) pourrait améliorer les résultats de façon significative. C'est pourquoi le programme que nous avons développé est conçu pour permettre l'implémentation, par une tierce personne, de nouvelles heuristiques d'ordonnancement de contraintes et de variables.

Enfin, l'étude de la recommandation et du calcul d'inférence par compilation de réseaux bayésiens fait actuellement l'objet d'un stage, qui se poursuivra en thèse. Les travaux que nous avons commencés et les idées que nous avons avancées sont actuellement développés et améliorés par cette personne, et feront l'objet d'articles à venir. Le compilateur SALADD est donc toujours utilisé et amélioré, que ce soit pour la compilation d'historiques en SLDD_+ et les outils adaptés à la recommandation, ou pour la compilation de réseaux bayésiens en $\text{SLDD}_\times$ pour le calcul d'inférence... et ça c'est chouette.

Bibliographie

Akers, S. (1978). Binary Decision Diagrams. *IEEE Transactions on Computers (TC)*, C-27:509–516.

Aloul, F. A., Markov, I. L. et 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)*.

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. et Marquis, P. (2002). Consistency restoration and explanations in dynamic CSPs - Application to configuration. *Artificial Intellligence Journal*, 135(1-2):199–234.

Amilhastre, J., Fargier, H., Niveau, A. et Pralet, C. (2014). Compiling CSPs : A complexity map of (non-deterministic) multivalued decision diagrams. *International Journal on Artificial Intelligence Tools*, 23(04).

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A. et Somenzi, F. (1993). Algebraic decision diagrams and their applications. *In Proceedings of the 12th International Conference Computer-Aided Design (ICCAD)*, pages 188–191.

Barwise, J. (1977). *Handbook of Mathematical Logic*. North-Holland.

Bessiere, C., Fargier, H. et Lecoutre, C. (2013). Global Inverse Consistency for Interactive Constraint Satisfaction. *In Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 159–174.

Beyer, D. (2008). *CrocoPat : a Tool for Simple and Efficient Relational Programming*. http ://www.cs.sfu.ca/ dbeyer/CrocoPat/.

BIERE, A. (2000). *ABCD : Armin Biere's Compact Decision Diagram BDD library*. http ://fmv.jku.at/abcd/. release 0.3.

BISTARELLI, S., MONTANARI, U., ROSSI, F., SCHIEX, T., VERFAILLIE, G. et FARGIER, H. (1999). Semiring-Based CSPs and Valued CSPs : Frameworks, Properties, and Comparison. *Constraints*, 4(3):199–240.

BRYANT, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers (TC)*, 38.8:677–691.

COOPER, G. F. (1990). The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks (Research Note). *Artificial Intelligence*, 42(2-3):393–405.

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

DARWICHE, A. (1999). Compiling Knowledge into Decomposable Negation Normal Form. *In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 284–289.

DARWICHE, A. (2001a). Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647.

DARWICHE, A. (2001b). On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34.

DARWICHE, A. (2003). A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)*, 50(3):280–305.

DARWICHE, A. (2004). New Advances in Compiling CNF into Decomposable Negation Normal Form. *In Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, pages 328–332.

DARWICHE, A. (2011). SDD : A new canonical representation of propositional knowledge bases. *In Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 819–826.

DARWICHE, A. et MARQUIS, P. (2002). A Knowledge Compilation Map. *Journal of Artificial Intelligence Research (JAIR)*, 17:229–264.

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

FARGIER, H. et MARQUIS, P. (2007). On Valued Negation Normal Form Formulas. *In Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 360–365.

FARGIER, H. et MARQUIS, P. (2009). Knowledge Compilation Properties of Trees-of-BDDs, Revisited. *In Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 772–777.

FARGIER, H., MARQUIS, P. et NIVEAU, A. (2013a). Towards a Knowledge Compilation Map for Heterogeneous Representation Languages. *In Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 877–883.

FARGIER, H., MARQUIS, P., NIVEAU, A. et SCHMIDT, N. (2014a). A Knowledge Compilation Map for Ordered Real-Valued Decision Diagrams. *In Proceedings of the 28th National Conference on Artificial Intelligence (AAAI)*, pages 1049–1055.

FARGIER, H., MARQUIS, P., NIVEAU, A. et SCHMIDT, N. (2014b). Carte de compilation des diagrammes de décision ordonnés à valeurs réelles. *Dans les actes des 8émes Journées de l'Intelligence Artificielle Fondamentale (JIAF)*, pages 127–136.

FARGIER, H., MARQUIS, P. et SCHMIDT, N. (2013b). Compacité pratique des diagrammes de décision valués : normalisation, heuristiques et expérimentations. *Dans les actes des 9èmes Journées Francophones de Programmation par Contraintes (JFPC)*, pages 123–132.

FARGIER, H., MARQUIS, P. et SCHMIDT, N. (2013c). Semiring Labelled Decision Diagrams, Revisited : Canonicity and Spatial Efficiency Issues. *In Proceedings of the 23rd Joint Conference on Artificial Intelligence (IJCAI)*, pages 884–890.

FARGIER, H., MARQUIS, P. et SCHMIDT, N. (2014c). Compacité pratique des diagrammes de décision valués. Normalisation, heuristiques et expérimentations. *Revue d'Intelligence Artificielle (RIA)*, 28(5):571–592.

FARGIER, H. et VILAREM, M. (2004). Compiling CSPs into Tree-Driven Automata for Interactive Solving. *Constraints*, 9(4):263–287.

GAREY, M. R. et JOHNSON, D. S. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman Publishers.

GERGOV, J. et MEINEL, C. (1994). Efficient Boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Transactions on Computers (TC)*, 43:1197–1209.

GOGIC, G., KAUTZ, H., PAPADIMITRIOU, C. et SELMAN, B. (1995). The Comparative Linguistics of Knowledge Representation. *In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 862–869.

HADZIC, T. (2004). A BDD-Based Approach to Interactive Configuration. *In Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, page 797.

HADZIC, T. et 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)*, pages 62–67.

HADZIC, T., JENSEN, R. M. et ANDERSEN, H. R. (2007). Calculating Valid Domains for BDD-Based Interactive Configuration. *The Computing Research Repository (CoRR) abs/0704.1394*.

HADZIC, T. et O'SULLIVAN, B. (2009). Uncovering functional dependencies in MDD-compiled product catalogues. *In Proceedings of the 3rd ACM Conference on Recommender Systems (RecSys)*, pages 377–380.

HOEY, J., ST-AUBIN, R., HU, A. et BOUTILIER, C. (1999). SPUDD : Stochastic Planning using Decision Diagrams. *In 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 279–288.

HUANG, J. et DARWICHE, A. (2004). Using DPLL for Efficient OBDD Construction. *In Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 157–172.

HUANG, J. et DARWICHE, A. (2005a). DPLL with a Trace : From SAT to Knowledge Compilation. *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 156–162.

HUANG, J. et DARWICHE, A. (2005b). On Compiling System Models for Faster and More Scalable Diagnosis. *In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, pages 300–306.

KISA, D., Van den BROECK, G., CHOI, A. et DARWICHE, A. (2014). Probabilistic sentential decision diagrams. *In Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

LAI, Y.-T. et SASTRY, S. (1992). Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification. *In Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC)*, pages 608–613.

LE BERRE, D. et PARRAIN, A. (2010). The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7(2-3):59–6.

LECOUTRE, C. et TABARY, S. (2006). Abscon 109 : a generic CSP solver. *Problème SAT : progrès et défis*, pages 243–267.

LECOUTRE, C. et TABARY, S. (2008). Abscon 112 : towards more robustness. *In Procedings of the 3rd International Constraint Solver Competition (CSC)*, pages 41–48.

LEE, C. (1959). Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38:985–999.

LIND-NIELSEN, J. (2002). *BuDDy : Binary Decision Diagrams Library Package*. http ://sourceforge.net/projects/buddy/. release 0.3.

MATEESCU, R. et DECHTER, R. (2006). Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs). *In Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 329–343.

MATEESCU, R. et DECHTER, R. (2007). AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Weighted Graphical Models. *In Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 276–284.

MATEESCU, R., DECHTER, R. et MARINESCU, R. (2008). AND/OR Multi-Valued Decision Diagrams (AOMDDs) for Graphical Models. *Journal of Artificial Intelligence Research (JAIR)*, 33:465–519.

MONTANARI, U. (1974). Networks of constraints : fundamental properties and applications to picture processing. *Information sciences*, 7:95–132.

NIVEAU, A., FARGIER, H., PRALET, C. et VERFAILLIE, G. (2010). Knowledge Compilation Using Interval Automata and Applications to Planning. *In Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*, pages 459–464.

PARGAMIN, B. (2003). Extending cluster tree compilation with non-boolean variables in product configuration : A tractable approach to preference-based configuration. *In Proceedings of the workshop on Configuration at*

the 18th International Joint Conference on Artificial Intelligence (IJCAI), volume 3.

PINKAS, G. (1991). Propositional non-monotonic reasoning and inconsistency in symmetric neural networks. *In Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 525–530.

PRALET, C., VERFAILLIE, G. et SCHIEX, T. (2006). Decision with uncertainties, feasibilities, and utilities : towards a unified algebraic framework. *In Proceedings of the 17th European Conference on Artificial Intelligence (ECAI)*, pages 427–431.

SANNER, S. et MCALLESTER, D. A. (2005). Affine Algebraic Decision Diagrams (AADDs) and their Application to Structured Probabilistic Inference. *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1384–1390.

SCHIEX, T., FARGIER, H. et VERFAILLIE, G. (1995). Valued Constraint Satisfaction Problems : Hard and Easy Problems. *In Proceedings of the 14th Joint Conference on Artificial Intelligence (IJCAI)*, pages 631–639.

SCUTARI, M. (2010). Learning Bayesian Network with the bnlearn **R** Package. *Journal of Statistical Software (JSS)*, 35(3):1–22.

SIELING, D. et WEGENER, I. (1993). NC-algorithms for operations on binary decision diagrams. *Parallel Processing Letters*, 3:3–12.

SINZ, C. (2002). Knowledge compilation for product configuration. *In Proceedings of the Workshop on Configuration at the 15th European Conference on Artificial Intelligence (ECAI)*, pages 23–26.

SOMENZI, F. (2005). *CUDD : Colorado University Decision Diagram package.* http ://vlsi.colorado.edu/ fabio/CUDD/. release 2.4.1.

SUBBARAYAN, S., BORDEAUX, L. et HAMADI, Y. (2007). Knowledge Compilation Properties of Tree-of-BDDs. *In Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)*, pages 502–507.

SZTIPANOVITS, J. et MISRA, A. (1996). Diagnosis of Discrete Event Systems Using Ordered Binary Decision Diagrams. *In Proceedings of the 7th International Workshop on Principles of Diagnosis.*

TAFERTSHOFER, P. et PEDRAM, M. (1997). Factored Edge-Valued Binary Decision Diagrams. *Formal Methods in System Design*, 10(2-3).

TARJAN, R. E. et YANNAKAKIS, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing (SICOMP)*, 13(3):566–579.

TORASSO, P. et TORTA, G. (2003). Computing Minimum-Cardinality Diagnoses Using OBDDs. *In Proceedings of Advances in Artificial Intelligence, 26th Annual German Conference on AI (KI)*, pages 224–238.

TSAMARDINOS, I., BROWN, L. E. et ALIFERIS, C. F. (2006). The max-min hill-climbing Bayesian network structure learning algorithm. *Machine learning*, 65(1):31–78.

VAHIDI, A. (2003). *JDD : a pure Java BDD and Z-BDD library.* http ://javaddlib.sourceforge.net/jdd/.

VEMPATY, N. R. (1992). Solving Constraint Satisfaction Problems Using Finite State Automata. *In Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*, pages 453–458.

WEGENER, I. (1987). *The complexity of Boolean functions.* Wiley-Teubner.

WHALEY, J. (2007). *JavaBDD : Java Library for Manipulating BDDs.* http ://javabdd.sourceforge.net/.

WILSON, N. (2005). Decision Diagrams for the Computation of Semiring Valuations. *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 331–336.

YATES, F. (1934). Contingency Tables Involving Small Numbers and the χ^2 Test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235.

Quatrième partie

Annexes

Annexe A

Définitions relatives au chapitre 2

GIC (*Global Inverse Consistency*) : Une valeur $d_i \in x$ d'un réseau de contraintes est Globalement Inversement Consistante (GIC) ssi il existe au moins une solution $\vec{x}$ tel que $x = d_i \subseteq \vec{x}$. Un réseau de contraintes est GIC ssi chaque valeur de chaque variable du réseau de contraintes est GIC.

Compacité : soient $L1$ et $L2$ deux langages sousensemble de NNF, $L1$ est au moins aussi compact que $L2$, noté $L1 \leqslant L2$ ssi il existe un polynôme p tel que pour toute formule $\alpha \in L2$, il existe une formule équivalente $\beta \in L1$ tel que $|\beta| \leqslant p(|\alpha|)$. ($|\alpha|$ et $|\beta|$ étant les tailles des formules α et β).

$L1$ est plus compact que $L2$, noté $L1 < L2$ ssi $L1 \leqslant L2$ et $L2 \nleqslant L1$.

$L1$ et $L2$ sont dit également compacts, noté $L1 \sim L2$ ssi $L1 \leqslant L2$ et $L2 \leqslant L1$.

NNF : Une NNF exprimant une formule logique f prend la forme d'un DAG, comportant une unique racine, où les nœuds terminaux sont étiquetés par soit un littéral, soit par les valeurs « $\top$ » et « $\bot$ », et les nœuds non terminaux sont étiquetés par l'opérateur de conjonction « $\wedge$ » ou de disjonction « $\vee$ », opérateurs non nécessairement binaires. La taille d'une formule f sous la forme NNF, notée $|f,|$ correspond au nombre de nœuds et au nombres d'arcs.

Déterminisme : Un NNF satisfait la propriété de déterminisme si pour toute disjonction C du NNF, chacune des sous formules de C sont logiquement contradictoires. Ainsi pour $C_1, ..., C_n$ les sous formules d'un nœud « $\vee$ », alors $\forall i, j$ tel que $i \neq j$, on a $C_i \wedge C_j \vDash \bot$.

Décomposabilité : Un NNF satisfait la propriété de décomposabilité si pour toute conjonction C du NNF, chacune des sous formules de C ne partagent aucune variable en commun. Ainsi pour $C_1, ..., C_n$ les sous formules d'un nœud « $\wedge$ », alors $\forall i, j$ tel que $i \neq j$, on a $Vars(C_i) \cap Vars(C_j) = \emptyset$.

DNNF : Le langage DNNF est le fragment de NNF satisfaisant la décomposabilité.
d-NNF : Le langage d-NNF est le fragment de NNF satisfaisant le déterminisme.
d-DNNF : Le langage d-DNNF est le fragment de NNF satisfaisant à la fois déterminisme et décomposabilité.

Uniformité : Un NNF satisfait la propriété d'uniformité (ou smoothness) si pour toute disjonction C du NNF, dans chacune des sous formules de C apparaissent les mêmes variables. Ainsi pour $C_1, ..., C_n$ les sous formules d'un nœud « $\vee$ », alors $Vars(C_i) = Vars(C_j)$.
Le langage s-NNF est le fragment de NNF satisfaisant la propriété d'uniformité. Il en va de même pour les s-DNNF, sd-NNF et sd-DNNF.

Aplatissement : Un NNF satisfait la propriété d'aplatissement (ou flatness) si sa hauteur maximale (c'est-à-dire le nombre de nœuds maximal que l'on peut rencontrer avant d'arriver à une feuille) est de 2.
Le langage f-NNF est le fragment de NNF satisfaisant la propriété d'aplatissement... et vous connaissez la suite...

BDD : Un diagramme de décision binaire (BDD, Binary Decision Diagrams) est un DAG représentant une fonction booléenne de variables booléennes. Il possède deux nœuds terminaux étiquetés $\top$ (« vrai ») et $\bot$ (« faux »). Leurs nœuds internes sont étiquetés par une variable booléenne et ont deux arcs sortants, étiquetés respectivement $\top$ et $\bot$.

FBDD : Les FBDD (free binary decision diagram) sont des BDD satisfaisant la propriété « read-once ». c'est-à-dire que quel que soit le chemin, de la racine à un nœud terminal, chaque variable ne pourra être rencontrée au maximum qu'une seule fois.

OBDD : Les OBDD (ordered binary decision diagram) sont des FBDD dans lesquels les variables sont ordonnées. C'est-à-dire que quel que soit le chemin, de la racine à un nœud terminal, les variables seront toujours rencontrées dans le même ordre.

OBDD$_\leq$: Les OBDD$_\leq$ sont des OBDD dans lesquels les variables sont ordonnées suivant un ordre « $\leq$ ».

CO, VA : un langage L satisfait **CO** (resp. **VA**) ssi il existe un algorithme polynômial en temps qui associe toute formule f dans L à 1 si f est cohérent (resp. valide) et à 0 si f ne l'est pas.

CE : un langage L satisfait **CE** ssi il existe un algorithme polynômial en temps qui associe toute formule f dans L et toute clause γ à 1 si $f \models \gamma$ et 0 si non.

IM : un langage L satisfait **IM** ssi il existe un algorithme polynômial en temps qui associe toute formule f dans L et toute clause γ à 1 si $\gamma \models f$ et 0 si non.

EQ, SE : un langage L satisfait **EQ** (resp. **SE**) ssi il existe un algorithme polynômial en temps qui associe tout couple de formule f et g dans L à 1 si $f \equiv g$ (resp. $f \models g$) et à 0 si non.

CT : un langage L satisfait **CT** ssi il existe un algorithme polynômial en temps qui donne à toute formule f dans L le nombre de modèles de f.

ME : un langage L satisfait **ME** ssi il existe un polynôme $p(.,.)$ et un algorithme capable de donner tous les modèles d'une formule f de L dans un temps $p(n, m)$, avec n la taille de f dans L et m le nombre de modèles de f.

CD : un langage L satisfait **CD** ssi il existe un algorithme polynômial en temps qui associe à toute formule f dans L et à toute clause γ une formule appartenant à L et équivalente à $f \mid \gamma$.

SFO, FO : un langage L satisfait **SFO** (resp. **FO**) ssi il existe un algorithme polynômial en temps qui associe à toute formule f dans L et à toute variable $x \in X$ (resp. à tout ensemble de variable $\mathcal{X} \in X$) une formule équivalente g privé de x (resp. privée de $\mathcal{X}$) tel que il existera toujours une affectation $\vec{x}$ de x (resp. $\vec{x}$ de $\mathcal{X}$) ou $g \models f(\vec{x})$.

$\wedge$BC, $\vee$BC : un langage L satisfait **$\wedge$BC** (resp. **$\vee$BC**) ssi il existe un algorithme polynômial en temps qui associe tout couple de formules f et g dans L à une formule de L équivalente à $f \wedge g$ (resp. $f \vee g$).

$\wedge$C, $\vee$C : un langage L satisfait **$\wedge$C** (resp. **$\vee$C**) ssi il existe un algorithme polynômial en temps qui associe tout ensemble fini de formules $f_1, \ldots, f_n$ dans L à une formule de L équivalente à $f_1 \wedge \cdots \wedge f_n$ (resp. $f_1 \vee \cdots \vee f_n$).

$\neg$C : un langage L satisfait **$\neg$C** ssi il existe un algorithme polynômial en temps qui associe toute formule f dans L à une formule de L équivalente à $\neg f$.

Vtree : Un *vtree* sur un ensemble de variables X est un arbre purement binaire dont les feuilles sont étiqueté par l'ensemble des variables de X, chaque variable de X apparaissant une unique fois parmi les feuilles d'un *vtree*.

VNNF : Un VNNF exprimant une formule logique f est un DAG comportant une unique racine. Il utilise une structure de valuation $\mathcal{E} = \langle E, \geq, OP \rangle$, les nœuds terminaux sont étiquetés par soit par une valeur de E soit par une fonction locale, et les nœuds non terminaux sont étiquetés par un opérateur de OP. La taille d'une formule f sous la forme VNNF, notée $|f|$ correspond au nombre de nœuds et au nombres d'arcs.

VCSP : un CSP valué est défini par un CSP classique (X, D, C) (variables, domaines, contraintes), une structure de valuation $S = (E, \otimes, \succ)$ et une application ϕ de C vers E. Pour un CSP valué (X, D, C, S, ϕ), une assignation A des variables $Y \subset X$, la valuation de A est définie par $f_A = \otimes[\phi(c)]|c \in C$ tel que A viole c.

AC : Un circuit arithmétique portant sur un ensemble de variables X est un DAG dont les nœuds terminaux sont étiquetés par des valeurs numériques ou des variables de X, et dont les autres nœuds sont étiquetés par les opérateurs $+$ ou $\times$.

EVBDD : Un Edge Valued BDD portant sur un ensemble de variables binaires X est un DAG dont les nœuds terminaux sont valués 0 et tout nœud non terminal n est défini par le 4-uplet $\langle var(n),\ child_{left}(n), child_{right}(n), \phi(n) \rangle$. Soit α l'EVBDD de racine n, β et γ les EVBDD de racine respectivement $child_{left}(n)$ et $child_{right}(n)$ et $var(n) = x \in X$, alors $f_\alpha = x(\phi_n + f_\beta) + (1-x)f_\gamma$.

Fonction locale : Une fonction locale [Pralet *et al.*, 2006] est une fonction portant sur un ensemble de variable $\mathcal{X} \subseteq X$ et à valeur dans E.

Généralement, une fonction locale sera soit une constante, soit une fonction portant sur une variable x pouvant prendre soit la valeur 1 si $x = d_i$ et 0 si $x \neq d_i$.

Annexe B

Modes d'utilisation du compilateur SALADD

Le compilateur SALADD peut être utilisé selon trois modes différents.

Le premier, conçu pour le démonstrateur BR4CP, se présente sous la forme d'une bibliothèque Java (.jar) implémentant les fonctions nécessaires au déroulement des différents protocoles de test.

Le deuxième, sous forme d'une archive, contient les sources et permet une utilisation du compilateur en ligne commande. Ce mode de fonctionnement permet la compilation d'un ou plusieurs fichiers (VCSP ou réseau bayésien), l'utilisation de l'ensemble des heuristiques présentés dans le corps de cette thèse, la sauvegarde des fichiers compilés, la possibilité de traduction vers chacun des langages ADD, SLDD et AADD, ainsi que la relecture d'un fichier sauvegardé.

Le troisième mode est destiné à un usage plus poussé. Il se présente sous la forme d'une bibliothèque Java (.jar). Cette bibliothèque possède toute les fonctions nécessaires à la compilation, la configuration de produit et la recommandation. Elle permet aussi d'implémenter ses propres heuristiques d'ordonnancement via l'implémentation d'une interface.

Les fichiers, bibliothèques, sources, ainsi que des descriptions des fonctions et des exemples d'utilisations peuvent être trouvés sur la page :
www.irit.fr/~Helene.Fargier/BR4CP/CompilateurSALADD.html

Annexe C

Preuves relatives au chapitre 4

Les pages qui vont suivrent contiennent les preuves de satisfaisabilité des langages ADD, SLDD_+, $\text{SLDD}_\times$ et AADD sur l'ensemble des requêtes et transformations introduites chapitre 4. Ces preuves sont extraites de la version étendue de l'article Fargier *et al.* [2014a].

Table 1: Results about basic queries, optimization, and γ-cutting, together with the number of the proposition (or corollary) proving each claim. $\surd$ means "satisfies", $\bullet$ means "does not satisfy", and $\circ$ means "does not satisfy unless $\mathsf{P} = \mathsf{NP}$".

Query	ADD	SLDD$_+$	SLDD$_\times$	AADD
EQ	$\surd$ C.7	$\surd$ C.7	$\surd$ C.7	$\surd$ C.7
SE	$\surd$ C.32	$\surd$ C.34	$\surd$ C.33	?
$\mathbf{OPT}_{\max}, \mathbf{OPT}_{\min}$	$\surd$ C.11	$\surd$ C.11	$\surd$ C.11	$\surd$ C.11
$\mathbf{CT}_{\max}, \mathbf{CT}_{\min}$	$\surd$ C.13	$\surd$ C.13	$\surd$ C.13	$\surd$ C.13
$\mathbf{ME}_{\max}, \mathbf{ME}_{\min}$	$\surd$ C.13	$\surd$ C.13	$\surd$ C.13	$\surd$ C.13
$\mathbf{MX}_{\max}, \mathbf{MX}_{\min}$	$\surd$ C.13	$\surd$ C.13	$\surd$ C.13	$\surd$ C.13
$\mathbf{CUT}_{\max}, \mathbf{CUT}_{\min}$	$\surd$ C.12	$\surd$ C.12	$\surd$ C.12	$\surd$ C.12
$\mathbf{VA}_{\sim\gamma}$	$\surd$ C.10	$\surd$ C.15	$\surd$ C.15	$\surd$ C.15
$\mathbf{VA}_{\succeq\gamma}, \mathbf{VA}_{\preceq\gamma}$	$\surd$ C.10	$\surd$ C.14	$\surd$ C.14	$\surd$ C.14
$\mathbf{CO}_{\sim\gamma}$	$\surd$ C.10	$\circ$ C.16	$\circ$ C.16	$\circ$ C.16
$\mathbf{CO}_{\succeq\gamma}, \mathbf{CO}_{\preceq\gamma}$	$\surd$ C.10	$\surd$ C.14	$\surd$ C.14	$\surd$ C.14
$\mathbf{ME}_{\sim\gamma}$	$\surd$ C.10	$\circ$ C.19	$\circ$ C.19	$\circ$ C.19
$\mathbf{ME}_{\succeq\gamma}, \mathbf{ME}_{\preceq\gamma}$	$\surd$ C.10	$\surd$ C.21	$\surd$ C.21	$\surd$ C.21
$\mathbf{MX}_{\sim\gamma}$	$\surd$ C.10	$\circ$ C.19	$\circ$ C.19	$\circ$ C.19
$\mathbf{MX}_{\succeq\gamma}, \mathbf{MX}_{\preceq\gamma}$	$\surd$ C.10	$\surd$ C.20	$\surd$ C.20	$\surd$ C.20
$\mathbf{CUT}_{\sim\gamma}$	$\surd$ C.9	$\bullet$ C.25	$\bullet$ C.25	$\bullet$ C.25
$\mathbf{CUT}_{\succeq\gamma}, \mathbf{CUT}_{\preceq\gamma}$	$\surd$ C.9	$\bullet$ C.26	$\bullet$ C.27	$\bullet$ C.27
$\mathbf{CT}_{\sim\gamma}$	$\surd$ C.10	$\circ$ C.19	$\circ$ C.19	$\circ$ C.19
$\mathbf{CT}_{\succeq\gamma}, \mathbf{CT}_{\preceq\gamma}$	$\surd$ C.10	$\circ$ C.24	$\circ$ C.24	$\circ$ C.24

Table 2: Results about transformations, together with the number of the proposition (or corollary) proving each claim. $\surd$ means "satisfies", $\bullet$ means "does not satisfy", and $\circ$ means "does not satisfy unless $\mathsf{P} = \mathsf{NP}$".

Transformation	ADD	SLDD$_+$	SLDD$_\times$	AADD
CD	$\surd$ C.8	$\surd$ C.8	$\surd$ C.8	$\surd$ C.8
max**BC**, min**BC**	$\surd$ C.31	$\bullet$ C.29	$\bullet$ C.29	$\bullet$ C.29
$+$**BC**	$\surd$ C.31	$\surd$ C.31	$\bullet$ C.30	$\bullet$ C.30
$\times$**BC**	$\surd$ C.31	$\bullet$ C.30	$\surd$ C.31	$\bullet$ C.30
max**C**, min**C**	$\bullet$ C.28	$\bullet$ C.28	$\bullet$ C.28	$\bullet$ C.28
$+$**C**	$\bullet$ C.28	$\bullet$ C.28	$\bullet$ C.28	$\bullet$ C.28
$\times$**C**	$\bullet$ C.28	$\bullet$ C.28	$\bullet$ C.28	$\bullet$ C.28
max**Elim**, min**Elim**	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36
$+$**Elim**	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36
$\times$**Elim**	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36
Smax**Elim**, **S**min**Elim**	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36
S$+$**Elim**	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36
S$\times$**Elim**	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36	$\bullet$ C.36
SBmax**Elim**, **SB**min**Elim**	$\surd$ C.37	$\bullet$ C.37	$\bullet$ C.37	$\bullet$ C.37
SB$+$**Elim**	$\surd$ C.37	$\surd$ C.37	$\bullet$ C.37	$\bullet$ C.37
SB$\times$**Elim**	$\surd$ C.37	$\bullet$ C.37	$\surd$ C.37	$\bullet$ C.37
max**Marg**, min**Marg**	$\surd$ C.39	$\surd$ C.39	$\surd$ C.39	$\surd$ C.39
$+$**Marg**	$\surd$ C.41	$\surd$ C.41	$\surd$ C.41	$\surd$ C.41
$\times$**Marg**	$\surd$ C.43	?	$\surd$ C.43	?

Proofs

Preliminaries

We first introduce a couple of technical propositions that are useful for proving the following results.

Proposition C.1. *Any* VDD *formula can be turned in polynomial time into an equivalent reduced formula in the same language.*

Proposition C.2. *There exists a linear algorithm transforming any* AADD *formula into an equivalent normalized* AADD *formula.*

Proposition C.3. *There exists a linear algorithm transforming any* SLDD$_+$ *(resp.* SLDD$_\times$*) formula into an equivalent normalized* SLDD$_+$ *(resp.* SLDD$_\times$*) formula.*

Proposition C.4. *For any* $\mathtt{L} \in \{\mathtt{ADD}, \mathtt{SLDD}_+, \mathtt{SLDD}_\times, \mathtt{AADD}\}$ *formula* α*,*

- *there is a unique (up to isomorphism) reduced and normalized* L *representation* α' *of* $f^{\mathtt{L}}_\alpha$*;*
- *no* L *representation of* $f^{\mathtt{L}}_\alpha$ *can be smaller than* $|\alpha'|$*.*

Proposition C.5. *Denoting* $\mathtt{L} \geq_\ell \mathtt{L}'$ *the fact that there exists a linear-time algorithm that associates with any* L *formula an equivalent* L′ *formula, the following results hold:*

- $\mathtt{ADD} \geq_\ell \mathtt{SLDD}_+ \geq_\ell \mathtt{AADD}$*;*
- $\mathtt{ADD} \geq_\ell \mathtt{SLDD}_\times \geq_\ell \mathtt{AADD}$*.*

Proposition C.6. *Any* ADD*,* SLDD$_+$*,* SLDD$_\times$*, or* AADD *representation of a function taking its values in* $\mathcal{V} = \{0, 1\}$ *can be translated in polynomial time into an* MDD *representation of the same function.*

Proof. We will show the result for AADD; the others can be inferred from it, since any formula in one of these languages can be transformed in linear time into an equivalent AADD formula (Proposition C.5). Note however that the result is actually obvious for the ADD language: when $\mathcal{V} = \mathcal{E} = \{0, 1\}$, $\mathtt{ADD} = \mathtt{MDD}$.

Any AADD formula can be normalized in linear time. Let α be a normalized AADD representation of a function taking its values in $\mathcal{V} = \{0, 1\}$. When α has no non-leaf node, it represents a constant function, so the translation to MDD is trivial. Suppose α contains at least one non-leaf node; we show by induction that all arcs that do not point to the leaf (including the offset-arc) have valuation $\langle 0, 1\rangle$.

Suppose that this is true for all arcs on some path from the offset-arc to a node N. Let $a \in \mathrm{Out}(N)$ be an arc that does not point to the leaf. Denoting M the destination node of a, the normalization condition also ensures that there exists one path from M to the leaf that has value 0, and another that has value 1. All in all, there exists a complete path of value q_a and another complete path of value $q_a + f_a$; these two values must be equal to 0 or 1. Since by the normalization condition, $f_a \neq 0$, the only possibility is that $q_a = 0$ and $f_a = 1$. This reasoning clearly applies to the offset arc, so by induction, all arcs not pointing to the leaf have valuation $\langle 0, 1\rangle$.

Now, let a be an arc pointing to the leaf. By the normalization condition, $f_a = 0$. Any complete path traversing a has value q_a (recall that along any path, all arcs except the last one have valuation $\langle 0, 1\rangle$), so q_a is either 0 or 1.

The conversion to MDD is thus direct: create a new leaf labeled with 1, redirect the $\langle 1, 0\rangle$ arcs to this new leaf, and remove all the arc valuations. □

Queries

Proposition C.7. ADD, SLDD$_+$, SLDD$_\times$, *and* AADD *satisfy* **EQ**.

Proof. This query is satisfied since each language L in $\{\texttt{ADD}, \texttt{SLDD}_+, \texttt{SLDD}_\times, \texttt{AADD}\}$ offers the canonicity property, i.e., any function has a unique reduced, normalized L representation given a fixed variable ordering (Proposition C.4), and since normalizing and reducing an L formula can be done in linear time (Proposition C.1 for the reduction, Proposition C.2 for the normalization of AADD, and Proposition C.3 for the normalization of SLDD). □

Proposition C.8. ADD, SLDD$_+$, SLDD$_\times$, *and* AADD *satisfy* **CD**.

Proof. We first show that single variable conditioning is in linear time on VDD formulæ. Assume that the conditioning of α by $x = v$ is asked for. For any node N labeled with x, the arcs in $\mathrm{Out}(N)$ corresponding to a value different from v must be discarded to get a VDD representation of the restriction $f_{\alpha,\langle x,v\rangle}$ of f_α. Because VDD formulæ are deterministic, there is only one arc a in $\mathrm{Out}(N)$ corresponding to value v.

In order to derive a VDD representation of $f_{\alpha,\langle x,v\rangle}$, each arc b in $\mathrm{In}(N)$ has to be redirected to the destination node of a and its label must be updated to take account of the removal of a. This update depends on the language considered: for ADD, there is nothing to do; for AADD, the label $\langle q_b, f_b\rangle$ of b must be replaced by $\langle q_b + f_b \times q_a, f_b \times f_a\rangle$; for SLDD$_+$ (resp. SLDD$_\times$), the label φ_b of b must be replaced by $\varphi_b + \varphi_a$ (resp. $\varphi_b \times \varphi_a$).

Accordingly, single variable conditioning can be achieved in time linear in the size of α, and the resulting formula has strictly fewer nodes and arcs that α (each node processing removes $|D_x| - 1$ arcs). Full conditioning simply amounts to iterating single variable conditioning for each conditioned variable, thus it runs in time polynomial in the input size. □

Proposition C.9. ADD *satisfies* $\mathbf{CUT}_{\sim\gamma}$, $\mathbf{CUT}_{\preceq\gamma}$, *and* $\mathbf{CUT}_{\succeq\gamma}$.

Proof. This is trivial, because the values taken by a function are listed as the leaves of its ADD representation. Hence, $CUT^{\sim\gamma}$ (resp. $CUT^{\preceq\gamma}$, $CUT^{\succeq\gamma}$) is the set of assignments the paths of which lead to a leaf with value $\varphi = \gamma$ (resp. $\varphi \leq \gamma$, $\varphi \geq \gamma$). Replacing the label of every such leaf by a and that of all other leaves by b, which can be done in time linear in the size of the formula, we obtain an ADD representation of the function g defined by $g(\vec{x}) = a$ if $\vec{x}$ is in the cut and $g(\vec{x}) = b$ otherwise. □

Corollary C.10. ADD *satisfies*

- $\mathbf{CO}_{\sim\gamma}$, $\mathbf{CO}_{\preceq\gamma}$, *and* $\mathbf{CO}_{\succeq\gamma}$;
- $\mathbf{VA}_{\sim\gamma}$, $\mathbf{VA}_{\preceq\gamma}$, *and* $\mathbf{VA}_{\succeq\gamma}$;
- $\mathbf{ME}_{\sim\gamma}$, $\mathbf{ME}_{\preceq\gamma}$, *and* $\mathbf{ME}_{\succeq\gamma}$;
- $\mathbf{MX}_{\sim\gamma}$, $\mathbf{MX}_{\preceq\gamma}$, *and* $\mathbf{MX}_{\succeq\gamma}$;
- $\mathbf{CT}_{\sim\gamma}$, $\mathbf{CT}_{\preceq\gamma}$, *and* $\mathbf{CT}_{\succeq\gamma}$.

Proof. This comes directly from the previous result: ADD satisfies $\mathbf{CUT}_{\sim\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$; resp. $\mathbf{CUT}_{\succeq\gamma}$), so an ADD formula α can be turned in polynomial time into an ADD formula β such that $f_\beta(\vec{x}) = 1$ if $\vec{x}$ is in $CUT^{\sim\gamma}(f_\alpha)$ (resp. $CUT^{\preceq\gamma}(f_\alpha)$; resp. $CUT^{\succeq\gamma}(f_\alpha)$) and 0 otherwise (we simply take $a = 1$ and $b = 0$). Thanks to Proposition C.6, β can be turned in polynomial time into an equivalent MDD formula. Since MDD satisfies the consistency, validity, model enumeration, model extraction, and model counting queries (Amilhastre et al. 2014), this proves that ADD satisfies $\mathbf{CO}_{\sim\gamma}$, $\mathbf{VA}_{\sim\gamma}$, $\mathbf{ME}_{\sim\gamma}$, $\mathbf{MX}_{\sim\gamma}$, and $\mathbf{CT}_{\sim\gamma}$ (resp. $\mathbf{CO}_{\preceq\gamma}$, $\mathbf{VA}_{\preceq\gamma}$, $\mathbf{ME}_{\preceq\gamma}$, $\mathbf{MX}_{\preceq\gamma}$, and $\mathbf{CT}_{\preceq\gamma}$; resp. $\mathbf{CO}_{\succeq\gamma}$, $\mathbf{VA}_{\succeq\gamma}$, $\mathbf{ME}_{\succeq\gamma}$, $\mathbf{MX}_{\succeq\gamma}$, and $\mathbf{CT}_{\succeq\gamma}$). □

Proposition C.11. ADD, SLDD$_+$, SLDD$_\times$, *and* AADD *satisfy* $\mathbf{OPT}_{\max}$ *and* $\mathbf{OPT}_{\min}$.

Proof. The satisfaction of $\mathbf{OPT}_{\max}$ and $\mathbf{OPT}_{\min}$ by AADD is due to Sanner and McAllester (2005); they showed that the maximal (resp. minimal) value reached by $f_\alpha^{\texttt{AADD}}$ is $q_0 + f_0$ (resp. q_0), where $\langle q_0, f_0\rangle$ is the offset of α (assumed to be normalized). Then the satisfaction of $\mathbf{OPT}_{\max}$ and $\mathbf{OPT}_{\min}$ by ADD,[1] SLDD$_+$, and SLDD$_\times$[2] is a consequence of Proposition C.5. □

Proposition C.12. ADD, SLDD$_+$, SLDD$_\times$, *and* AADD *satisfy* $\mathbf{CUT}_{\max}$ *and* $\mathbf{CUT}_{\min}$.

Proof. Let us recall that, for any normalized AADD representation α, the assignments $\vec{x}$ such that $f_\alpha^{\texttt{AADD}}(\vec{x})$ is maximal correspond to the paths following only the arcs e such that $q_e + f_e = 1$ (they always exist when α is normalized). It is then possible to get an ADD representation of $CUT^{\max}(f_\alpha^{\texttt{AADD}})$ by replacing the leaf with a new one labeled with a (the a-node) and creating a second leaf labeled with b (the b-node): the arcs such that $q_e + f_e \neq 1$ are then redirected to the b-node (this is done in time linear in the number of arcs). The structure is cleaned by recursively deleting nodes without incoming arcs; this cleaning process is also in linear time. The procedure is thus a polynomial algorithm transforming an AADD formula α into an ADD representation of $CUT^{\max}(f_\alpha^{\texttt{AADD}})$. This algorithm allow us to conclude about $\mathbf{CUT}_{\max}$ for all four languages, thanks to Proposition C.5: since ADD $\geq_\ell$ AADD, AADD satisfies $\mathbf{CUT}_{\max}$ (we can transform the resulting ADD formula into AADD) and ADD too (we can transform the initial formula into AADD, and keep the resulting ADD formula); SLDD$_+$ (resp. SLDD$_\times$) satisfies $\mathbf{CUT}_{\max}$ because SLDD$_+$ $\geq_\ell$ AADD (resp. SLDD$_\times$ $\geq_\ell$ AADD), which allows us to "prepare" the formula to use the algorithm, and ADD $\geq_\ell$ SLDD$_+$ (resp. ADD $\geq_\ell$ SLDD$_\times$), which allows us to translate the result into the targeted language.

[1] The satisfaction of $\mathbf{OPT}_{\max}$ and $\mathbf{OPT}_{\min}$ by ADD is obvious (just explore the terminal nodes to get the maximal or minimal value labeling one of them).

[2] For SLDD$_+$ and SLDD$_\times$, our result also coheres with that of Wilson (2005), despite the fact that $(\mathbb{R}^+, \min, +)$ and $(\mathbb{R}^+, \max, +)$ are not semirings. Indeed, this does not matter, the important point being the addition-is-max (or addition-is-min) assumption. See the original paper from Wilson (2005) for details.

The proof is similar for $\mathbf{CUT}_{\min}$, since for any normalized AADD representation α, the assignments $\vec{x}$ such that $f_\alpha^{\mathtt{AADD}}(\vec{x})$ is minimal correspond to the paths following only the arcs e such that $q_e = 0$ (they always exist when α is normalized). The leaf becomes the a-node and the arcs e such that $q_e \neq 0$ are redirected to the b-node. □

Corollary C.13. $\mathtt{ADD}$, $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *satisfy* $\mathbf{ME}_{\max}$, $\mathbf{ME}_{\min}$, $\mathbf{MX}_{\max}$, $\mathbf{MX}_{\min}$, $\mathbf{CT}_{\max}$, $\mathbf{CT}_{\min}$.

Proof. This comes directly from the previous result: for any $\mathtt{L} \in \{\mathtt{ADD}, \mathtt{SLDD}_\times, \mathtt{SLDD}_+, \mathtt{AADD}\}$, L satisfies $\mathbf{CUT}_{\max}$ (resp. $\mathbf{CUT}_{\min}$), so an L formula can be turned in polynomial time into an L formula α such that $f_\alpha^{\mathtt{L}}(\vec{x}) = 1$ if $\vec{x}$ is a maximal (resp. minimal) value of $f_\alpha^{\mathtt{L}}$, and 0 otherwise (we simply take $a = 1$ and $b = 0$). Thanks to Proposition C.6, α can be turned in polynomial time into an equivalent MDD formula. Since MDD satisfies the model enumeration, model counting, and model extraction queries, this proves that L satisfies $\mathbf{ME}_{\max}$, $\mathbf{CT}_{\max}$, and $\mathbf{MX}_{\max}$ (resp. $\mathbf{ME}_{\min}$, $\mathbf{CT}_{\min}$, and $\mathbf{MX}_{\min}$). □

Proposition C.14. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *satisfy* $\mathbf{CO}_{\succeq\gamma}$, $\mathbf{VA}_{\succeq\gamma}$, $\mathbf{CO}_{\preceq\gamma}$, $\mathbf{VA}_{\preceq\gamma}$.

Proof.
- To determine whether there exists an $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma$ (resp. $f_\alpha^{\mathtt{L}}(\vec{x}) \leq \gamma$), it is sufficient to compute the maximal (resp. minimal) value v^* (resp. v_*) reached by $f_\alpha^{\mathtt{L}}$; this can be done in polynomial time since $\mathbf{OPT}_{\max}$ (resp. $\mathbf{OPT}_{\min}$) is satisfied by all three languages (Proposition C.11). Then it is enough to compare this value to γ, since $\exists\vec{x}, f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma \iff v^* \geq \gamma$ (resp. $\exists\vec{x}, f_\alpha^{\mathtt{L}}(\vec{x}) \leq \gamma \iff v_* \leq \gamma$).
- To determine whether all the $\vec{x}$ are such that $f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma$ (resp. $f_\alpha^{\mathtt{L}}(\vec{x}) \leq \gamma$), it is sufficient to compute the minimal (resp. maximal) value v_* (resp. v^*) reached by $f_\alpha^{\mathtt{L}}$; this can be done in polynomial time since $\mathbf{OPT}_{\min}$ (resp. $\mathbf{OPT}_{\max}$) is satisfied by all three languages. Then it is enough to compare this value to γ, since it holds that $\forall\vec{x}, f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma \iff v_* \geq \gamma$ (resp. $\forall\vec{x} f_\alpha^{\mathtt{L}}(\vec{x}) \leq \gamma \iff v^* \leq \gamma$). □

Corollary C.15. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *satisfy* $\mathbf{VA}_{\sim\gamma}$.

Proof. Checking whether all the $\vec{x}$ are such that $f_\alpha^{\mathtt{L}}(\vec{x}) = \gamma$ is equivalent to checking whether for each $\vec{x}$, both $f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma$ and $f_\alpha^{\mathtt{L}}(\vec{x}) \leq \gamma$ hold. Since all three languages satisfy both $\mathbf{VA}_{\succeq\gamma}$ and $\mathbf{VA}_{\preceq\gamma}$ (Proposition C.14), they also satisfy $\mathbf{VA}_{\sim\gamma}$. Note that this proof does not work for $\mathbf{CO}_{\sim\gamma}$. □

Proposition C.16. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{CO}_{\sim\gamma}$, *unless* $\mathsf{P} = \mathsf{NP}$.

Proof. We first show that $\mathtt{SLDD}_+$ does not satisfy $\mathbf{CO}_{\sim\gamma}$unless $\mathsf{P} = \mathsf{NP}$. This follows from Hadzic and Andersen (2006, Theorem 3), by reduction of SUBSET SUM (given a set of integers $E = \{i_1, \ldots, i_n\}$ of cardinality n and an integer k, is there a subset of E summing to k?) which is NP-complete (Garey and Johnson 1979) even if all the integers in E are positive.

Let us associate with E in polynomial time an $\mathtt{SLDD}_+$ formula α_E over a set $\mathcal{X} = \{x_1, \ldots, x_n\}$ of n Boolean variables as follows (see Figure 1): the root of α_E is labeled with x_1, α_E contains n internal nodes respectively labeled with $x_1, \ldots, x_n$ plus one leaf; for each i_j of E, we build two arcs e_j and e'_j in α_E from the x_j node to the x_{j+1} node (or to the leaf when $j = n$): e_j corresponds to $x_j = 0$ and $\varphi(e_j) = 0$; e'_j corresponds to $x_j = 1$ and $\varphi(e'_j) = i_j$. This construction is done in time linear in $|E|$. Each $\vec{x}$ (and thus each path in α) is in bijection with a subset of E and $f_\alpha^{\mathtt{SLDD}_+}(\vec{x})$ is equal to the sum of the elements in this subset.

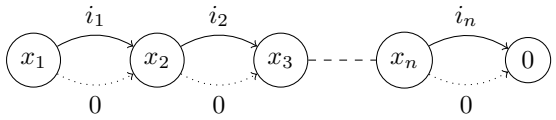


Figure 1: The $\mathtt{SLDD}_+$ formula α_E

If $\mathtt{SLDD}_+$ satisfied $\mathbf{CO}_{\sim\gamma}$, then it would be possible to check in polynomial time whether there exists a subset of E summing to k – i.e., to solve the SUBSET SUM problem. Hence $\mathtt{SLDD}_+$ does not satisfy $\mathbf{CO}_{\sim\gamma}$, unless $\mathsf{P} = \mathsf{NP}$.

Now, a similar construction holds for the $\mathtt{SLDD}_\times$ language, by considering the SUBSET PRODUCT problem (given a set of integers $E = \{i_1, \ldots, i_n\}$ of cardinality n and an integer k, is there a subset whose product is k?) which is NP-complete as well (Garey and Johnson 1979).

Finally, if $\mathtt{AADD}$ satisfied $\mathbf{CO}_{\sim\gamma}$, then $\mathtt{SLDD}_+$ also would, since any $\mathtt{SLDD}_+$ formula can be turned into an equivalent $\mathtt{AADD}$ formula in linear time (Proposition C.5). However, this is not the case unless $\mathsf{P} = \mathsf{NP}$, hence $\mathtt{AADD}$ does not satisfy $\mathbf{CO}_{\sim\gamma}$ unless $\mathsf{P} = \mathsf{NP}$. □

Proposition C.17. *Let* L *be a representation language over* $\mathcal{X}$ *w.r.t.* $\mathcal{V}$, *where* $\mathcal{V}$ *is totally ordered by a relation* $\succeq$. *If* L *satisfies* $\mathbf{MX}_{\sim\gamma}$ *or* $\mathbf{CT}_{\sim\gamma}$, *then* L *satisfies* $\mathbf{CO}_{\sim\gamma}$.

Proof. If L satisfies $\mathbf{MX}_{\sim\gamma}$, then we can find in polynomial time an $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$ or the information that there is no such assignment, so we can check in polynomial time whether there exists an $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$. This shows that $\mathbf{MX}_{\sim\gamma}$ implies $\mathbf{CO}_{\sim\gamma}$. Now, it is obvious that counting the number of $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$ indicates whether there exists an $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$ (the answer is yes if and only if this number is positive); hence if L satisfies $\mathbf{CT}_{\sim\gamma}$, it also satisfies $\mathbf{CO}_{\sim\gamma}$. □

Proposition C.18. *Let* L *be a representation language over* $\mathcal{X}$ *w.r.t.* $\mathcal{V}$, *where* $\mathcal{V}$ *is totally ordered by a relation* $\succeq$. *If* L *satisfies* $\mathbf{ME}_{\sim\gamma}$, *then* L *satisfies* $\mathbf{CO}_{\sim\gamma}$.

Proof. If L satisfies $\mathbf{ME}_{\sim\gamma}$, then there exists a polynomial $p(\cdot,\cdot)$ and an algorithm A taking as input an L formula α and listing every element of $CUT^{\sim\gamma}(f_\alpha^{\mathtt{L}})$ in time bounded by $p(|\alpha|, n)$, where n is the number of elements in $CUT^{\sim\gamma}(f_\alpha^{\mathtt{L}})$.

Now, let A' be the algorithm that, taking as input an L formula α, simulates the execution of A on input α, and stops

after at most $p(|\alpha|, 0)$ steps. The result it returns depends on what happened during the simulation. There are three possible cases:

(i) if A stopped without returning anything, then A' returns 0;

(ii) if A stopped after having returned at least one assignment, then A' returns 1;

(iii) if A did not stop by itself, then A' returns 1.

Obviously A' runs in time polynomial in the size of α. Now, it is not hard to see that A' returns 1 if there exists an assignment $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$, and 0 otherwise.

Indeed, suppose that no such assignment exists. Then by definition, $CUT^{\sim\gamma}(f_\alpha^{\mathtt{L}})$ is empty, so A stops without having returned anything after at most $p(|\alpha|, 0)$ steps. This is case i: A' returns 0.

Now, suppose that there exists an assignment $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$. This means that $CUT^{\sim\gamma}(f_\alpha^{\mathtt{L}})$ is not empty; let us denote n its cardinal. The algorithm A returns n assignments and stops after at most $p(|\alpha|, n)$ steps. Nothing prevents $p(|\alpha|, n)$ to be lower than $p(|\alpha|, 0)$: this corresponds to case ii, in which A' returns 1. Finally, in the (probably less unusual) case when $p(|\alpha|, n) > p(|\alpha|, 0)$, A' has to interrupt the simulation of A; this is case iii, in which A' returns 1.

In conclusion, if $\mathtt{L}$ satisfies $\mathbf{ME}_{\sim\gamma}$, then there exists a polynomial-time algorithm deciding whether there exists an assignment $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \sim \gamma$, that is, $\mathtt{L}$ satisfies $\mathbf{CO}_{\sim\gamma}$. □

Proposition C.19. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{MX}_{\sim\gamma}$, $\mathbf{CT}_{\sim\gamma}$, *or* $\mathbf{ME}_{\sim\gamma}$, *unless* $\mathsf{P} = \mathsf{NP}$.

Proof. From Proposition C.16, none of these languages satisfies $\mathbf{CO}_{\sim\gamma}$ unless $\mathsf{P} = \mathsf{NP}$. This implies, from Propositions C.17 and C.18, that they cannot satisfy $\mathbf{MX}_{\sim\gamma}$, $\mathbf{CT}_{\sim\gamma}$, or $\mathbf{ME}_{\sim\gamma}$, unless $\mathsf{P} = \mathsf{NP}$. □

Proposition C.20. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *satisfy* $\mathbf{MX}_{\succeq\gamma}$ *and* $\mathbf{MX}_{\preceq\gamma}$.

Proof. To get an assignment $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma$, simply check whether γ is strictly greater than the maximal value v^*, which can be obtained in polynomial time since all the languages satisfy $\mathbf{OPT}_{\max}$ (Proposition C.11). If so, we know that there is no assignment $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma$. In the remaining case (i.e., when $v^* \geq \gamma$), the optimal assignments $\vec{x}^*$ are such that $f_\alpha^{\mathtt{L}}(\vec{x}^*) = v^* \geq \gamma$; since all three languages satisfy $\mathbf{MX}_{\max}$ (Proposition C.13), one can get such an $\vec{x}^*$ in polynomial time. Since by construction $\vec{x}^* \in CUT^{\succeq\gamma}$, this proves that all three languages satisfy $\mathbf{MX}_{\succeq\gamma}$. We prove in the same way that they satisfy $\mathbf{MX}_{\preceq\gamma}$, as a consequence of the satisfaction of $\mathbf{OPT}_{\min}$ (Proposition C.11) and $\mathbf{MX}_{\min}$ (Proposition C.13). □

Proposition C.21. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *satisfy* $\mathbf{ME}_{\succeq\gamma}$ *and* $\mathbf{ME}_{\preceq\gamma}$.

Proof. We prove the result for $\mathtt{AADD}$ first, using the basic idea that at any node N of a normalized $\mathtt{AADD}$ formula, there is at least one path to the leaf of valuation 1 and one path to the leaf of valuation 0. A path reaching N provides an offset, gathered from its arcs, say $\langle p, q\rangle$, so the cheapest path traversing N has valuation p and the most expensive one has valuation $p + q$.

This allows us to traverse the graph in reverse topological order, only developing a node if it yields at least one element of the cut (e.g., if we want the elements of $CUT^{\succeq\gamma}$, node N is only developed if $p + q \geq \gamma$). We thus get the entire list of elements in the cut by a tree search algorithm that is backtrack-free, and thus polynomial in the number of leaves reached, which is exactly the number of assignments $\vec{x}$ such that $f_\alpha^{\mathtt{L}}(\vec{x}) \geq \gamma$ (resp. $f_\alpha^{\mathtt{L}}(\vec{x}) \leq \gamma$). A recursive implementation of this procedure is detailed in Algorithm 1 (the top call is supposed to pass $\vec{x} = \vec{\varnothing}$). Note that the assignments listed are not always complete, in the sense that they only feature variables encountered on the corresponding path. It is straightforward to extend each resulting partial assignment $\vec{z}$ into the full list of complete assignments of which $\vec{z}$ is a restriction.

Algorithm 1: $\mathsf{EnumModelsAADD}(\alpha, \vec{x}, \mathcal{R}, \gamma)$

input : an $\mathtt{AADD}$ formula α, of root N, with offset $\langle q_0, f_0\rangle$; a current assignment $\vec{x}$; a relation $\mathcal{R} \in \{\leq, \geq\}$, and a threshold $\gamma \in \mathbb{R}^+$
output: the list of assignments $\vec{x} \cdot \vec{y}$ such that $f_\alpha^{\mathtt{AADD}}(\vec{x} \cdot \vec{y})\ \mathcal{R}\ \gamma$

```
if R = ≤ then
    if q0 > γ then
        return
else
    if q0 + f0 < γ then
        return
if N is the leaf then
    print x
    return
let y := Var(N)
foreach arc a going out of N do
    let y = ⟨y, v(a)⟩
    let M be the destination node of a
    let α' be the AADD formula rooted at the destination
    node of a, with offset ⟨q0 + f0 × qa, f0 × fa⟩
    EnumModelsAADD(α, x · y), R, γ)
```

This proves that $\mathtt{AADD}$ satisfies $\mathbf{ME}_{\succeq\gamma}$ and $\mathbf{ME}_{\preceq\gamma}$; we can deduce from Proposition C.5 that $\mathtt{SLDD}_+$ and $\mathtt{SLDD}_\times$ also satisfy these two queries. □

Lemma C.22. *Let* $\otimes \in \{+, \times\}$. *There exists a polynomial-time algorithm mapping any* $\mathtt{SLDD}_\otimes$ *formula* α *and every set of variables* $X \supseteq \mathrm{Var}(\alpha)$ *to an equivalent* $\mathtt{SLDD}_\otimes$ *formula in which each path mentions all the variables in* X.

Proof. Let α be an $\mathtt{SLDD}_\otimes$ formula, in which variables are

ordered with respect to $\lhd$. Let $X = \{x_1, \ldots, x_n\}$ be a set of variables containing all variables in $\mathrm{Var}(\alpha)$; without loss of generality, let us assume that $x_1 \lhd \cdots \lhd x_n$.

What we want is to build an $\mathtt{L}$ formula α' in which every path from the root to the leaf is of the form $\langle a_1, \ldots, a_n \rangle$ where each a_i (with $i \in \{1, \ldots, n-1\}$) is an arc from a node labeled with x_i to a node labeled with x_{i+1}, and a_n is an arc from a node labeled with x_n to the leaf node.

This property can be easily guaranteed in polynomial time: for each arc a of α from a node N_i labeled with x_i ($i \in \{1, \ldots, n-1\}$) to a node M such that either M is the leaf or $\mathrm{Var}(M) = x_{i+j}$ with $j > 1$,

1. add $j-1$ new nodes $N_{i+1}, \ldots, N_{i+j-1}$ respectively labeled $x_{i+1}, \ldots, x_{i+j-1}$;
2. redirect a to N_{i+1}; and
3. for each of the new nodes N_k ($k \in \{i, \ldots, i+j-1\}$), for each value $d \in D_{x_k}$, add an arc $a_{k,d}$ from N_k to N_{k+1} (conveniently considering N_{i+j} to be M) with $v(a_{k,d}) = d$ and $\varphi(a_{k,d})$ is the neutral element of $\otimes$ (0 for $+$ and 1 for $\times$). □

Lemma C.23. *There exists a polynomial algorithm mapping any* $\mathtt{SLDD}_+$ *formula* α *of maximal value* v^* *and any constant* $K \in \mathbb{R}^+$ *such that* $K \geq v^*$*, to an* $\mathtt{SLDD}_+$ *representation of the function* $g = K - f_\alpha^{\mathtt{SLDD}_+}$.

Proof. This is not hard to prove, but care must be taken to always remain in the fragment (negative arc valuations are forbidden by the definitions).

We first compute v^*, the maximal value taken by $f_\alpha^{\mathtt{SLDD}_+}$; this can be done in polynomial time, since $\mathtt{SLDD}_+$ satisfies $\mathbf{OPT}_{\max}$ (Proposition C.11). Then we modify α so that all of its paths mentions all the variables in $\mathrm{Var}(\alpha)$ (Lemma C.22 states that it can be done in polynomial time), and every arc label φ (including the offset) is replaced by $K - \varphi$, which is guaranteed to be in $\mathbb{R}^+$ since $K \geq v^*$. We call α' the modified formula.

It should be clear that for any $\vec{x}$, $f_{\alpha'}^{\mathtt{SLDD}_+}(\vec{x}) = K \times (1 + |\mathrm{Var}(\alpha)|) - f_\alpha(\vec{x})$, since along each path, we added K as many times as there are arcs, plus one time for the offset. Applying the normalization procedure on α' (in linear time, thanks to Proposition C.3), we get an $\mathtt{SLDD}_+$ formula α'' with offset $K \times (1 + |\mathrm{Var}(\alpha)|) - v^*$: indeed, the offset of a normalized $\mathtt{SLDD}_+$ formula is the minimal value taken by the function it represents (each node has at least one 0-valued outgoing arc).

We now simply have to substract $K \times |\mathrm{Var}(\alpha)|$ to the offset of α'' to obtain an $\mathtt{SLDD}_+$ representation of the function $g = K - f_\alpha^{\mathtt{SLDD}_+}$. □

Proposition C.24. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{CT}_{\succeq\gamma}$ *or* $\mathbf{CT}_{\preceq\gamma}$ *unless* $\mathsf{P} = \mathsf{NP}$.

Proof. We first show that $\mathtt{SLDD}_+$ cannot satisfy $\mathbf{CT}_{\succeq\gamma}$ unless $\mathsf{P} = \mathsf{NP}$. Let us suppose it is the case; then we can compute in polynomial time the number $M_{\geq\gamma}$ of assignments $\vec{x}$ such that $f_\alpha^{\mathtt{SLDD}_+}(\vec{x}) \geq \gamma$ (i.e., compute $|CUT^{\succeq\gamma}(f_\alpha^{\mathtt{SLDD}_+})|$). It is moreover always possible to compute the maximal value v^* reached by $f_\alpha^{\mathtt{SLDD}_+}$ ($\mathtt{SLDD}_+$ satisfies $\mathbf{OPT}_{\max}$, Proposition C.11) and to get a representation in the same language of the function $g_\alpha = v^* - f_\alpha^{\mathtt{SLDD}_+}$ (thanks to Lemma C.23). Since we suppose that $\mathbf{CT}_{\succeq\gamma}$ is satisfied, then it is possible to compute in polynomial time the number M of assignments $\vec{x}$ such that $v^* - f_\alpha^{\mathtt{SLDD}_+}(\vec{x}) \geq v^* - \gamma$, taking $v^* - \gamma$ as the threshold. We can then compute the number $M_{>\gamma} = |D_1 \times \cdots \times D_n| - M$ of assignments $\vec{x}$ such that $v^* - f_\alpha^{\mathtt{SLDD}_+}(\vec{x}) < v^* - \gamma$: by construction, $M_{>\gamma}$ is equal to the number of assignments $\vec{x}$ such that $f_\alpha^{\mathtt{SLDD}_+}(\vec{x}) > \gamma$.

Since we know that there are $M_{\geq\gamma}$ assignments $\vec{x}$ such that $f_\alpha^{\mathtt{SLDD}_+}(\vec{x}) \geq \gamma$, we can deduce that there are precisely $M_{\geq\gamma} - M_{>\gamma}$ assignments $\vec{x}$ such that $f_\alpha^{\mathtt{SLDD}_+}(\vec{x}) = \gamma$.

We have proved that the satisfaction of $\mathbf{CT}_{\succeq\gamma}$ implies that of $\mathbf{CT}_{\sim\gamma}$; since $\mathtt{SLDD}_+$ does not satisfy $\mathbf{CT}_{\sim\gamma}$ unless $\mathsf{P} = \mathsf{NP}$ (Proposition C.19), we conclude that $\mathtt{SLDD}_+$ does not satisfy $\mathbf{CT}_{\succeq\gamma}$ unless $\mathsf{P} = \mathsf{NP}$.

We prove that $\mathtt{SLDD}_+$ does not satisfy $\mathbf{CT}_{\preceq\gamma}$ unless $\mathsf{P} = \mathsf{NP}$ in a similar way: the satisfaction of $\mathbf{CT}_{\preceq\gamma}$ would allow us to compute $M_{\leq\gamma}$ and $M_{<\gamma}$ using the same technique, hence it would imply the satisfaction of $\mathbf{CT}_{\sim\gamma}$.

These two results imply that $\mathtt{SLDD}_\times$ cannot satisfy $\mathbf{CT}_{\preceq\gamma}$ or $\mathbf{CT}_{\succeq\gamma}$ unless $\mathsf{P} = \mathsf{NP}$. Indeed, an $\mathtt{SLDD}_+$ representation of a function g can be transformed in polynomial time into an $\mathtt{SLDD}_\times$ representation of the function $h = 2^g$, simply by replacing the label of the leaf by 1 and the value φ of each arc (including the offset) by 2^φ. If $\mathtt{SLDD}_\times$ satisfied $\mathbf{CT}_{\preceq\gamma}$ (resp. $\mathbf{CT}_{\succeq\gamma}$), we could obtain in polynomial time the number of assignments $\vec{x}$ such that $2^{g(\vec{x})} \leq 2^\gamma$ (resp. $2^{g(\vec{x})} \geq 2^\gamma$), which is the same number as $CUT^{\preceq\gamma}(g)$ (resp. $CUT^{\succeq\gamma}(g)$). Thus the satisfaction of $\mathbf{CT}_{\preceq\gamma}$ (resp. $\mathbf{CT}_{\succeq\gamma}$) on $\mathtt{SLDD}_\times$ would imply its satisfaction on $\mathtt{SLDD}_+$, yet we have just proved that it is not satisfied by $\mathtt{SLDD}_+$ unless $\mathsf{P} = \mathsf{NP}$.

Finally, $\mathtt{AADD}$ does not satisfy $\mathbf{CT}_{\succeq\gamma}$ (resp. $\mathbf{CT}_{\preceq\gamma}$), because this would imply that $\mathtt{SLDD}_+$ also satisfies it, since $\mathtt{SLDD}_+ \geq_\ell \mathtt{AADD}$ (Proposition C.5). □

Proposition C.25. $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{CUT}_{\sim\gamma}$.

Proof. We consider a set $\mathcal{X} = \{y_1, \ldots, y_n, z_1, \ldots, z_n\}$ of $2n$ Boolean variables, and $\lhd$ the total order on $\mathcal{X}$ defined by $y_1 \lhd \cdots \lhd y_n \lhd z_1 \lhd \cdots \lhd z_n$. Let f^{sum} and f^{prod} be the functions defined by $f^{\mathrm{sum}}(\vec{y} \cdot \vec{z}) = \sum_{i=1}^n y_i 2^{n-i} + \sum_{i=1}^n z_i 2^{n-i}$, and $f^{\mathrm{prod}}(\vec{y} \cdot \vec{z}) = 2^{f^{\mathrm{sum}}(\vec{y} \cdot \vec{z})}$. Figure 2 shows how f^{sum} (resp. f^{prod}) can be represented as an $\mathtt{SLDD}_+$ (resp. $\mathtt{SLDD}_\times$) formula ordered by $\lhd$ containing only $2n+1$ nodes and $2n$ arcs. Since $\mathtt{SLDD}_+ \geq_\ell \mathtt{AADD}$ (Proposition C.5), f^{sum} also has a linear-sized $\mathtt{AADD}$ representation.

Let f^{eq} be the Boolean function defined by $f^{\mathrm{eq}}(\vec{y} \cdot \vec{z}) = 1$ if $f^{\mathrm{sum}}(\vec{y} \cdot \vec{z}) = 2^n$, and 0 otherwise. Note that f^{eq} also equals 1 if and only if $f^{\mathrm{prod}}(\vec{y} \cdot \vec{z}) = 2^{2^n}$.

We now show that there is no polynomial-sized representation of f^{eq} as an $\mathtt{OBDD}_\lhd$ formula. The proof basically relies on a result by Sieling and Wegener (1993), showing that for any Boolean function f over $\mathcal{X}$, if the number of restrictions on $\{y_1, \ldots, y_n\}$ which are distinct and depends on z_1

is equal to m, then any $\texttt{OBDD}_\lhd$ formula representing f contains at least m nodes labeled with z_1. We show that it is the case for f^{eq}.

Clearly enough, every assignment $\overrightarrow{y}$ over $\{y_1, \ldots, y_n\}$ (resp. $\overrightarrow{z}$ over $\{z_1, \ldots, z_n\}$) is associated in a bijective way with a natural number $\mathrm{N}(\overrightarrow{y}) = \sum_{i=1}^{n} y_i 2^{n-i}$ (resp. $\mathrm{N}(\overrightarrow{z}) = \sum_{i=1}^{n} z_i 2^{n-i}$) which belongs to $[0, 2^n - 1]$. In particular, there are 2^n distinct assignments over $\{y_1, \ldots, y_n\}$. Consider now two distinct assignments $\overrightarrow{y}$ and $\overrightarrow{y}'$ over $\{y_1, \ldots, y_n\}$. The two restrictions $f^{\text{eq}}_{\overrightarrow{y}}$ and $f^{\text{eq}}_{\overrightarrow{y}'}$ are distinct, since they have distinct support sets (i.e., the set of assignments over $\{z_1, \ldots, z_n\}$ that makes the restriction equal to 1), namely $\{\, \overrightarrow{z} \mid \mathrm{N}(\overrightarrow{y}) + \mathrm{N}(\overrightarrow{z}) = 2^n \,\}$ and $\{\, \overrightarrow{z} \mid \mathrm{N}(\overrightarrow{y}') + \mathrm{N}(\overrightarrow{z}) = 2^n \,\}$, which are singletons (there is exactly one $\overrightarrow{z}$ for each $\overrightarrow{y}$).

Furthermore, for each assignment $\overrightarrow{y}$ over $\{y_1, \ldots, y_n\}$, $f^{\text{eq}}_{\overrightarrow{y}}$ depends on z_1, i.e., $f^{\text{eq}}_{\overrightarrow{y}\cdot\langle z_1,0\rangle} \neq f^{\text{eq}}_{\overrightarrow{y}\cdot\langle z_1,1\rangle}$: this is obvious since the support set of $f^{\text{eq}}_{\overrightarrow{y}}$ is a singleton (one of the two functions always returns 0, whereas the other one returns 1 for exactly one assignment).

Consequently, according to the abovementioned result, any $\texttt{OBDD}_\lhd$ formula representing f^{eq} contains at least $2^n - 1$ nodes labeled with z_1: there can be no polynomial-sized $\texttt{OBDD}_\lhd$ representation of f^{eq}.

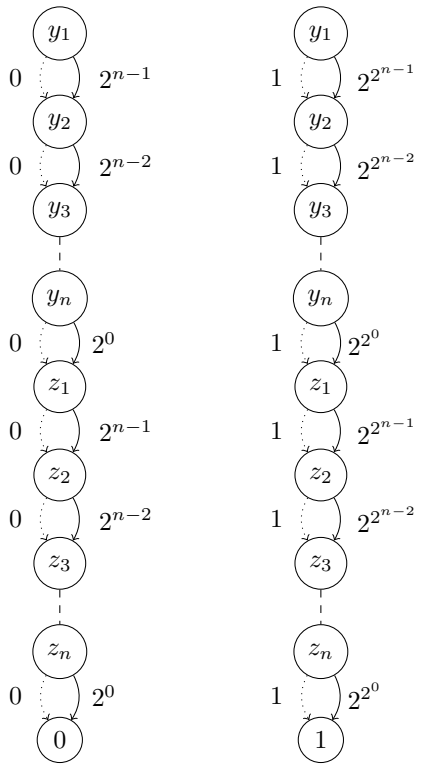


Figure 2: Left: An $\texttt{SLDD}_+$ representation of f^{sum}. Right: An $\texttt{SLDD}_\times$ representation of f^{prod}.

Now, note that the model set of f^{eq} is exactly $CUT^{\sim\gamma}(f^{\text{sum}}) = CUT^{\sim\gamma}(f^{\text{prod}})$ with $\gamma = 2^n$. This means that if $\texttt{SLDD}_+$ (resp. $\texttt{SLDD}_\times$, $\texttt{AADD}$) satisfied $\mathbf{CUT}_{\sim\gamma}$, we could get in time polynomial in n an $\texttt{SLDD}_+$ (resp. $\texttt{SLDD}_\times$, $\texttt{AADD}$) representation of f^{eq} with $a = 1$ and $b = 0$, that we could in turn translate in polynomial time into an $\texttt{OBDD}_\lhd$ formula (using Proposition C.6, and remarking that an MDD over Boolean variables is an OBDD). By contradiction, $\texttt{SLDD}_+$, $\texttt{SLDD}_\times$, and $\texttt{AADD}$ do not satisfy $\mathbf{CUT}_{\sim\gamma}$. □

Proposition C.26. $\texttt{SLDD}_+$ *does not satisfy* $\mathbf{CUT}_{\succeq\gamma}$ *or* $\mathbf{CUT}_{\preceq\gamma}$.

Proof. The first step is to show that $\texttt{SLDD}_+$ satisfies $\mathbf{CUT}_{\succeq\gamma}$ if and only if it satisfies $\mathbf{CUT}_{\preceq\gamma}$. Let $a, b \in \mathbb{R}^+$ such that $a > b$, and let $\gamma \in \mathbb{R}^+$.

Suppose $\texttt{SLDD}_+$ satisfies $\mathbf{CUT}_{\succeq\gamma}$. Computing the maximal value v^* of $f_\alpha^{\texttt{SLDD}_+}$, which can be done in polynomial time (Proposition C.11), and choosing a constant $K \geq \max(v^*, \gamma)$, Lemma C.23 states that we can build in polynomial time an $\texttt{SLDD}_+$ representation α' of the function $K - f_\alpha^{\texttt{SLDD}_+}$. Now, applying the $\mathbf{CUT}_{\succeq\gamma}$ transformation to α' with a threshold of $K - \gamma$, which by hypothesis can be done in polynomial time, we get an $\texttt{SLDD}_+$ representation of the function g defined by $g(\overrightarrow{x}) = a$ if $f_{\alpha'}^{\texttt{SLDD}_+}(\overrightarrow{x}) \geq K - \gamma$, i.e., if $f_\alpha^{\texttt{SLDD}_+}(\overrightarrow{x}) \leq \gamma$, and $g(\overrightarrow{x}) = b$ otherwise: $\mathbf{CUT}_{\preceq\gamma}$ is satisfied. This proves that if $\texttt{SLDD}_+$ satisfies $\mathbf{CUT}_{\succeq\gamma}$, it also satisfies $\mathbf{CUT}_{\preceq\gamma}$. The same reasoning can be used to show that if it satisfies $\mathbf{CUT}_{\preceq\gamma}$, it also satisfies $\mathbf{CUT}_{\succeq\gamma}$. Hence, the satisfaction of either of the two implies the satisfaction of both. We now show that it also implies the satisfaction of $\mathbf{CUT}_{\sim\gamma}$.

Let $\gamma \in \mathbb{R}^+$ and let α be an $\texttt{SLDD}_+$ formula. We apply the $\mathbf{CUT}_{\succeq\gamma}$ and $\mathbf{CUT}_{\preceq\gamma}$ transformations on α, with $a = 1$ and $b = 0$. The resulting $\texttt{SLDD}_+$ formulæ can be turned in polynomial time into ordered MDDs (Proposition C.6), and since $\texttt{MDD}$ satisfies $\wedge\mathbf{BC}$ (Amilhastre et al. 2014), we can obtain in polynomial time an MDD the models of which are exactly the assignments that are in $CUT^{\succeq\gamma}(f_\alpha) \cap CUT^{\preceq\gamma}(f_\alpha)$, i.e., an MDD representing $CUT^{\sim\gamma}(f_\alpha)$. We can replace the 0-leaf by some $b \in \mathbb{R}^+$ and the 1-leaf by some $a \in \mathbb{R}^+$ such that $a > b$: we get an $\texttt{ADD}$ representation of the function h defined by $h(\overrightarrow{x}) = a$ if $\overrightarrow{x} \in CUT^{\sim\gamma}(f_\alpha)$ and $h(\overrightarrow{x}) = b$ otherwise. Since $\texttt{ADD} \geq_\ell \texttt{SLDD}_+$ (Proposition C.5), this means that $\texttt{SLDD}_+$ satisfies $\mathbf{CUT}_{\sim\gamma}$.

All in all, if $\texttt{SLDD}_+$ satisfies $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$), it also satisfies $\mathbf{CUT}_{\preceq\gamma}$ (resp. $\mathbf{CUT}_{\succeq\gamma}$), which in turn implies that it satisfies $\mathbf{CUT}_{\sim\gamma}$, yet we have shown that it does not (Proposition C.25). Hence $\texttt{SLDD}_+$ does not satisfy $\mathbf{CUT}_{\succeq\gamma}$ or $\mathbf{CUT}_{\preceq\gamma}$. □

Corollary C.27. $\texttt{SLDD}_\times$ *and* $\texttt{AADD}$ *do not satisfy* $\mathbf{CUT}_{\succeq\gamma}$ *or* $\mathbf{CUT}_{\preceq\gamma}$.

Proof. We show that if $\texttt{SLDD}_\times$ satisfied $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$), $\texttt{SLDD}_+$ would also satisfy it. Indeed, an $\texttt{SLDD}_+$ representation α of a function f can be transformed in polynomial time into an $\texttt{SLDD}_\times$ representation α' of the function $g = 2^f$, simply by replacing the label of the leaf by 1 and the value φ of each arc (including the offset) by 2^φ. Suppose

$\texttt{SLDD}_\times$ satisfies $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$): taking $a = 1$ and $b = 0$, we could obtain in polynomial time an $\texttt{SLDD}_\times$ representing $CUT^{\succeq\gamma}(f)$ (resp. $CUT^{\preceq\gamma}(f)$), by taking the cut of α' w.r.t. the threshold 2^γ. We can transform the result into an MDD, which only takes polynomial time (Proposition C.6), then replace the 0-leaf by some $b \in \mathbb{R}^+$ and the 1-leaf by some $a \in \mathbb{R}^+$ such that $a > b$: we get an ADD representation of the function h defined by $h(\vec{x}) = a$ if $\vec{x}$ is in $CUT^{\succeq\gamma}(f)$ (resp. $CUT^{\preceq\gamma}(f)$) and $h(\vec{x}) = b$ otherwise. Since $\texttt{ADD} \geq_\ell \texttt{SLDD}_+$ (Proposition C.5), this means that $\texttt{SLDD}_+$ satisfies $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$), yet we have shown that it does not (Proposition C.26). Consequently, $\texttt{SLDD}_\times$ cannot satisfy $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$).

Now, using a reasoning similar to that of the previous paragraph, we show that if AADD satisfied $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$), $\texttt{SLDD}_+$ also would. Indeed, an $\texttt{SLDD}_+$ formula can be turned in polynomial time into an AADD formula (Proposition C.5); then we could apply $\mathbf{CUT}_{\succeq\gamma}$ (resp. $\mathbf{CUT}_{\preceq\gamma}$) on values 0 and 1, so that the resulting AADD formula can be turned in polynomial time into an MDD (Proposition C.6); then the leaves of the MDD can be re-labeled with any $a, b \in \mathbb{R}^+$, and the resulting ADD formula be turned in polynomial time into an $\texttt{SLDD}_+$ (Proposition C.5). □

Combinations

Proposition C.28. *ADD, $\texttt{SLDD}_+$, $\texttt{SLDD}_\times$, and AADD do not satisfy $+\mathbf{C}$, $\times\mathbf{C}$, $\max\mathbf{C}$, or $\min\mathbf{C}$.*

Proof. The proofs rely on results in the Boolean case: Wegener (1987) has shown that there exists a family of formulæ Σ_n over $\{x_1, \ldots, x_n\}$ where each Σ_n has a number of prime implicates cubic in n (hence each Σ_n has a polynomial-sized CNF representation) but every OBDD representation of Σ_n has a size exponential in n. We use the Σ_n family in all the following proofs.

ADD **does not satisfy** $+\mathbf{C}$**.** For each n, for each of the clauses δ_i of Σ_n, let α_i be an ADD formula of size linear in the size of δ_i (hence linear in n) representing a term equivalent to $\neg\delta_i$, i.e., such that for each assignment $\vec{x}$ over $\{x_1, \ldots, x_n\}$, $f^{\texttt{ADD}}_{\alpha_i}(\vec{x}) = 0$ if $\vec{x}$ satisfies δ_i, $f^{\texttt{ADD}}_{\alpha_i}(\vec{x}) = 1$ otherwise. Suppose that ADD satisfied $+\mathbf{C}$. In this case, it would be possible to compute in time polynomial in n an ADD representation of the function that associates with each assignment $\vec{x}$ the number of clauses of Σ_n violated by $\vec{x}$. Since ADD satisfies $\mathbf{CUT}_{\min}$ (Proposition C.12), we could then compute from it in polynomial time an ADD representation with $a = 1$ and $b = 0$ of the set of assignments $\vec{x}$ such that $f_{\Sigma_n}(\vec{x}) = 1$ if $\vec{x}$ does not violate any clause, $f_{\Sigma_n}(\vec{x}) = 0$ otherwise. By construction, this ADD representation is also an OBDD representation of Σ_n. This would contradict the fact that every OBDD representation of Σ_n has exponential size.

ADD **does not satisfy** $\max\mathbf{C}$**.** Consider the ADD formulæ α_i as in the proof for the $+\mathbf{C}$ case. If ADD satisfied $\max\mathbf{C}$, then it would be possible to compute in time polynomial in n an ADD representation of the function f_{Σ_n} that associates with each assignment $\vec{x}$ the value $\max_i f^{\texttt{ADD}}_{\alpha_i}(\vec{x})$. By construction, this value is equal to 1 if Σ is violated by $\vec{x}$ and is equal to 0 otherwise. Hence the ADD representation of the function f_{Σ_n} also is an OBDD representation of $\neg\Sigma_n$. An OBDD representation of Σ_n could be easily obtained from it by labeling the 0-leaf with 1 and the 1-leaf with 0. This would contradict the fact that every OBDD representation of Σ_n has exponential size.

ADD **does not satisfy** $\times\mathbf{C}$**.** For each of the clauses δ_i of Σ_n, let α_i be an ADD representation of δ_i. Each α_i can be easily generated in time linear in the size of δ_i, hence in time linear in n. If ADD satisfied $\times\mathbf{C}$, it would be possible to compute in time polynomial in n an ADD representation of the function f_{Σ_n} that associates with each assignment $\vec{x}$ the value $\prod_i f^{\texttt{ADD}}_{\alpha_i}(\vec{x})$. This value is equal to 1 if Σ_n is satisfied by $\vec{x}$ and to 0 otherwise. Hence it is an OBDD representation of Σ_n, contradiction.

ADD **does not satisfy** $\min\mathbf{C}$**.** For each of the clauses δ_i of Σ_n, let α_i be an ADD representation of δ_i. Each α_i can be easily generated in time linear in the size of δ_i, hence in time linear in n. If ADD satisfied $\min\mathbf{C}$, then it would be possible to compute in time polynomial in n an ADD representation of the function f_{Σ_n} that associates with each assignment $\vec{x}$ the value $\min_i f^{\texttt{ADD}}_{\alpha_i}(\vec{x})$,. This value is equal to 0 if Σ is violated by $\vec{x}$ and is equal to 1 otherwise. Hence it is an OBDD representation of Σ_n, contradiction.

$\texttt{SLDD}_+$**,** $\texttt{SLDD}_\times$**, and** AADD **do not satisfy** $+\mathbf{C}$**,** $\times\mathbf{C}$**,** $\max\mathbf{C}$**, or** $\min\mathbf{C}$**.** Let $\texttt{L} \in \{\texttt{SLDD}_+, \texttt{SLDD}_\times, \texttt{AADD}\}$. Any ADD formula can be transformed in linear time into an equivalent L formula (Proposition C.5). Furthermore, L satisfies $\mathbf{CUT}_{\min}$ (Proposition C.12). Finally, for each transformation, each one of the functions f_{Σ_n} considered in the above proofs takes its values in $\{0, 1\}$. Hence, thanks to Proposition C.6, each of its L representations (as computed as in the above proofs) could be turned in polynomial time into an equivalent MDD representation (which would be an OBDD representation, since every variable is a Boolean one). Once again, this would contradict the fact that every OBDD representation of Σ_n has exponential size. □

Proposition C.29. $\texttt{SLDD}_+$*,* $\texttt{SLDD}_\times$*, and* AADD *do not satisfy* $\max\mathbf{BC}$ *or* $\min\mathbf{BC}$*.*

Proof. We first show that the satisfaction of $\max\mathbf{BC}$ implies that of $\mathbf{CUT}_{\preceq\gamma}$.

Let $\gamma \in \mathcal{V}$; we can easily generate in constant time an ADD representation of the constant function f_γ defined by $\forall\vec{x}, f_\gamma(\vec{x}) = \gamma$. Given Proposition C.5, we can also generate in constant time an $\texttt{SLDD}_+$ (resp. $\texttt{SLDD}_\times$, resp. AADD) representation of f_γ.

Suppose that $\texttt{L} \in \{\texttt{SLDD}_+, \texttt{SLDD}_\times, \texttt{AADD}\}$ satisfied $\max\mathbf{BC}$. Then for any L formula α, it would be possible to build in time polynomial in the size of α an L representation β of the function $g = \max(f_\alpha, f_\gamma)$. Because each of the three languages satisfies $\mathbf{OPT}_{\min}$ (Proposition C.11), we could compute in polynomial time the minimum value v_* reached by g. If $v_* > \gamma$, then $\forall\vec{x}, f_\alpha(\vec{x}) > \gamma$. Accordingly, $CUT^{\preceq\gamma}(f_\alpha) = \varnothing$. So the cut $f_{\preceq\gamma}$ on $\{b, c\}$ is the constant function such that $\forall\vec{x}, f_{\preceq\gamma}(\vec{x}) = c$, which can be represented in constant time as an L formula with only one node.

The case when $v_* \leq \gamma$ is more complicated, but we show that it is possible to build in polynomial time an L representation of $f_{\preceq\gamma}$, by redirecting the arcs not taking part in valuations equal to γ.

First, we can compute in polynomial time, for any arc a of an L formula α, the cost $mincost(a)$ of the cheapest assignment $\vec{x}$ such that $p(\vec{x})$ contains a.

This is clear when L is $\mathtt{SLDD}_+$ or $\mathtt{SLDD}_\times$: using a shortest path algorithm, we can compute in polynomial time, for any node N, the cost $min_{\text{out}}(N)$ of the cheapest path from N to the leaf, and the cost $min_{\text{in}}(N)$ of the cheapest path from the root to N. Then for any arc a in α, denoting M and N its source and destination nodes, $mincost(a) = min_{\text{in}}(M) \otimes \varphi_a \otimes min_{\text{out}}(N)$, where $\otimes$ is $+$ for $\mathtt{SLDD}_+$ and $\times$ for $\mathtt{SLDD}_\times$.

In the case of AADD, this is less direct. We use the fact that for any node N in a normalized AADD formula, there always exist a path from N to the leaf with valuation 0. This allows the computation of $mincost(a)$ for each arc a in one traversal of the graph from the root to the sink in topological order. The procedure is described in Algorithm 2; the idea is to compute an "offset" for each node N, representing the aggregation of the valuation pairs of each arc along the cheapest path from the root to N.

Algorithm 2: MinCostArcsAADD(α)

input : an AADD formula α, of root R, with offset $\langle q_0, f_0 \rangle$
output: the value of $mincost(a)$ for each arc a in α

```
foreach node N of α in reverse topological ordering do
    if N is the root node then
        let q_min(N) := q_0
        let f_min(N) := f_0
    else
        let a_min := arg min_{a∈In(N)} mincost(a)
        let q_min(N) := q_min(a_min)
        let f_min(N) := f_min(a_min)
    foreach arc a going out of N do
        let q_min(a) := q_min(N) + f_min(N) * q_a
        let f_min(N) := f_min(N) * f_a
        let mincost(a) = q_min(a) + f_min(a)
return { ⟨a, mincost(a)⟩ | a arc in α }
```

It is thus possible, for any $\mathtt{L} \in \{\mathtt{SLDD}_+, \mathtt{SLDD}_\times, \mathtt{AADD}\}$, to obtain in polynomial time the value $mincost(a)$ for any arc a in the L representation α' of the function $g = \max(f_\alpha, f_\gamma)$. With a simple transformation, we can turn α' into an L representation of the cut $f_{\preceq\gamma}$ on $\{b, c\}$.

The procedure is as follows: label the leaf with b; add a new c-labeled leaf; redirect every arc a such that $mincost(a) > \gamma$ to the new c leaf; remove all arc valuations. The result is an ADD formula, which represents $f_{\preceq\gamma}$. Indeed, let $\vec{x} \in CUT^{\preceq\gamma}(f_\alpha)$; clearly, $g(\vec{x}) = \gamma$, so each arc a along the path $p(\vec{x})$ in α' is such that $mincost(a) = \gamma$: the path is still the same in β, and leads to the b-leaf.

Conversely, consider an assignment $\vec{x}$ such that $f_\beta(\vec{x}) = b$; we show that $\vec{x} \in CUT^{\preceq\gamma}(f_\alpha)$. Let us denote as $p_\beta(\vec{x})$ the path corresponding to $\vec{x}$ in β, and as $p_{\alpha'}(\vec{x})$ the corresponding path in α'. It is clear that these two paths contain the same arcs, since $p_\beta(\vec{x})$ leads to the b-leaf (no arc has been modified). Denoting $\langle a_1, \ldots, a_n \rangle$ the sequence of arcs along these paths, we know by construction that for each a_i, $mincost(a_i) = \gamma$. We show by induction that for all $k \in \{1, n\}$, there exists a path starting with $a_1, \ldots, a_k$ of valuation γ.

This is trivial for $k = 1$, since $mincost(a_1) = \gamma$. Suppose the hypothesis holds for some $k \in \{1, n-1\}$; we show it holds for $k+1$. Let p_k be the path in α' from the source node of a_{k+1} to the leaf such that $\varphi(\langle a_1, \ldots, a_k \rangle \cdot p_k) = \gamma$, where $\varphi(p)$ denotes the valuation of some path p. Since $mincost(a_{k+1}) = \gamma$, there exists a path of valuation γ containing a_{k+1}: let us denote p_{in} the part before a_{k+1} and p_{k+1} the part after a_{k+1}.

By construction, each path p in α' verifies $\varphi(p) \geq \gamma$ (recall that α' represents $\max(f_\alpha, f_\gamma)$). Hence, we know that the following inequalities hold:

$$\varphi(p_{\text{in}} \cdot p_k) \geq \gamma$$
$$\varphi(\langle a_1, \ldots, a_k, a_{k+1} \rangle \cdot p_{k+1}) \geq \gamma$$

(it can be easily checked that the two paths are legal paths). Now, since $\varphi(p_{\text{in}} \cdot \langle a_{k+1} \rangle \cdot p_{k+1}) = \gamma$, we can deduce from the first inequation that $\varphi(p_k) \geq \varphi(\langle a_{k+1} \rangle \cdot p_{k+1})$, of which we can deduce that $\varphi(\langle a_1, \ldots, a_k \rangle \cdot p_k) \geq \varphi(\langle a_1, \ldots, a_k, a_{k+1} \rangle \cdot p_{k+1})$ (it can be checked that these deductions are valid for any $\mathtt{L} \in \{\mathtt{SLDD}_+, \mathtt{SLDD}_\times, \mathtt{AADD}\}$).

Since $\varphi(\langle a_1, \ldots, a_k \rangle \cdot p_k) = \gamma$, it holds that $\gamma \geq \varphi(\langle a_1, \ldots, a_k, a_{k+1} \rangle \cdot p_{k+1})$; combining this result with the second inequation, we get that $\varphi(\langle a_1, \ldots, a_k, a_{k+1} \rangle \cdot p_{k+1}) = \gamma$: the proposition holds for $k+1$, therefore by induction it holds for all $k \in \{1, n\}$. The fact that it holds for $k = n$ implies that $\varphi(\langle a_1, \ldots, a_n \rangle) = \gamma$: $f_{\alpha'}(\vec{x}) = \gamma$, so $\vec{x} \in CUT^{\preceq\gamma}(f_\alpha)$.

This proves that β is an ADD representation of $f_{\preceq\gamma}$ on $\{b, c\}$; it could be transformed in linear time into an equivalent L representation (Proposition C.5). Hence, for $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, or AADD, the satisfaction of $\max\mathbf{BC}$ implies that of $\mathbf{CUT}_{\preceq\gamma}$; since these languages do not satisfy $\mathbf{CUT}_{\preceq\gamma}$, they cannot satisfy $\max\mathbf{BC}$.

It can be shown in a similar way that for these three languages, the satisfaction of $\min\mathbf{BC}$ implies that of $\mathbf{CUT}_{\succeq\gamma}$, considering the min-combination of α with an L representation of the constant function f_γ. Since $\mathbf{CUT}_{\succeq\gamma}$ is not satisfied by $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, and AADD, we conclude in the same way that these languages do not satisfy $\min\mathbf{BC}$. □

Proposition C.30.

- $\mathtt{SLDD}_+$ *and* AADD *do not satisfy* $\times\mathbf{BC}$*;*
- $\mathtt{SLDD}_\times$ *and* AADD *do not satisfy* $+\mathbf{BC}$*.*

Proof. The proofs of the two items work in a similar way; we first consider the case of $\times\mathbf{BC}$ on $\mathtt{SLDD}_+$ and AADD.

Let f be the function of n Boolean variables (with $n \geq 3$) defined by $\forall\vec{x}$, $f(\vec{x}) = \sum_{i=0}^{n-1} x_i \cdot 2^i$ (this is the function associating a n-bit vector with its corresponding integer), and g the function of n Boolean variables defined by $\forall\vec{x}$,

$g(\vec{x}) = 2^n + \sum_{i=0}^{n-1}(1-x_i)\cdot 2^i$. It holds that $\forall \vec{x}$, $f(\vec{x}) + g(\vec{x}) = 2^{n+1} - 1$.

We consider the variable ordering given by $x_0 \lhd x_1 \lhd \cdots \lhd x_{n-1}$.

Each of the two functions f and g has an $\mathtt{SLDD}_+$ representation with $n+1$ nodes and $2n$ arcs (one node per variable, two arcs per node). In the one of f (see Figure 3), the node at level x_i has one arc with label 1 and valuation 2^i and the other with label 0 and valuation 0. In the one of g (see Figure 4), the arc valuations are inverted at each level, and the root is associated with an offset $\varphi_0 = 2^n$.

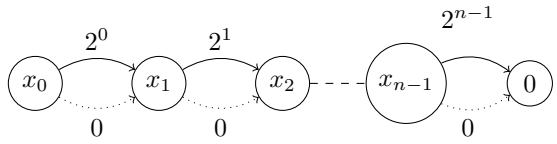


Figure 3: A $\mathtt{SLDD}_+$ representation of f

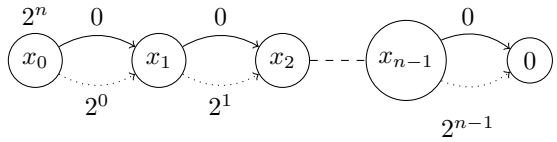


Figure 4: A $\mathtt{SLDD}_+$ representation of g

Let us consider the following ADD representation α of $h = f \times g$. There are 2^n paths in α; let us denote them $p_0, \ldots, p_{2^n-1}$, with p_k being the one that corresponds to the assignment $\vec{x}$ such that $f(\vec{x}) = k$. The path p_k leads to a leaf of value $k(2^{n+1} - k - 1)$, that we denote as h_k. The sequence $\langle h_k \rangle_{0 \le k < 2^n}$ is strictly increasing (since $k \le 2^{n-1}$, $\delta_k = h_k - h_{k-1} = 2(2^n - k)$, $\delta_k > 4$ for $0 \le k \le 2^{n-1}$), so all leaves are labeled with a different value. Now, let us build a AADD representation of h from α. Following the procedure outlined in the proof of Proposition C.5, we start with the ADD formula α, and add a neutral valuation ($\langle 0, 1 \rangle$) to every arc except for those pointing to a leaf L, which receive the value $\langle \varphi(L), 0 \rangle$. Stated otherwise, the last arc of each path p_k is valued $\langle h_k, 0 \rangle$. Then the leaves are merged into a unique leaf, and the AADD formula obtained is normalized (Proposition C.2). Let us suppose the algorithm is treating some node at level x_{n-1} (the level closest to the leaf); it is traversed by two paths, say p_{k-1} and p_k. We have $q_{min} = h_{k-1}$ and $range = h_k - h_{k-1} = \delta_k$, which is greater than 0, so the h_{k-1} arc receives value $\langle 0, 0 \rangle$, the h_k arc receives value $\langle 1, 0 \rangle$, and the unique incoming arc receives value $\langle h_{k-1}, h_k - h_{k-1} \rangle$. Clearly, all nodes at level x_{n-1} are isomorphic but none of them is redundant.

Now, let us suppose the algorithm is treating some node N at level x_{n-2}. It is traversed by four paths, say p_{4j}, p_{4j+1}, p_{4j+2}, and p_{4j+3}. Its two outgoing arcs are respectively valued with $\langle h_{4j}, \delta_{4j+1} \rangle$ and $\langle h_{4j+2}, \delta_{4j+3} \rangle$. In this case, $q_{min} = h_{4j}$ and $range = h_{4j+3} - h_{4j}$. Since $range > 0$, the 0-labeled outgoing arc of N receives the value $\langle 0, \delta_{4j+1}/range \rangle$. Let us denote v_j the multiplicative constant of this valuation: $v_j = \delta_{4j+1}/(h_{4j+3} - h_{4j})$.

It is tedious, yet not difficult, to check that the sequence $\langle v_j \rangle_{0 \le j \le \lfloor (2^n-1)/4 \rfloor}$ is strictly increasing (recall that we suppose $n \ge 3$). Hence, the 0-labeled outgoing arcs of all nodes at level x_{n-2} are all valued with a distinct multiplicative constant: none of them can be isomorphic to another one. Furthermore, the 1-labeled outgoing arc of N receives the value $\langle (h_{4j+2} - h_{4j})/(h_{4j+3} - h_{4j}), (h_{4j+3} - h_{4j+2})/(h_{4j+3} - h_{4j}) \rangle$. Since $h_{4j+2} - h_{4j} > 0$, this label differs from the one of the 0-labeled outgoing arc of N, hence N is not redundant.

Consequently, after applying the normalization and the reduction procedures, the resulting AADD formula has at least $\lfloor (2^n - 1)/4 \rfloor + 1 = 2^{n-2}$ distinct nodes at level x_{n-2}. Recall that no AADD formula can be strictly smaller than an equivalent reduced and normalized AADD formula (Proposition C.4). All in all, we have the following:

(i) f and g have $\mathtt{SLDD}_+$ representations of size polynomial in n;
(ii) the size of the smallest AADD representation of $f \times g$ is exponential in n.

Since any $\mathtt{SLDD}_+$ representation can be transformed into an AADD formula in linear time (Proposition C.5):

(a) f and g have an AADD representation of size polynomial in n, because of (i);
(b) the size of the smallest $\mathtt{SLDD}_+$ representation of $f \times g$ is exponential in n, because of (ii).

To sum up, $\mathtt{SLDD}_+$ does not satisfy $\times\mathbf{BC}$ (thanks to (i) and (b)), and AADD does not satisfy $\times\mathbf{BC}$ either (thanks to (ii) and (a)).

Now, for $+\mathbf{BC}$ on $\mathtt{SLDD}_\times$ and AADD, the proof is very similar to the previous one, using different functions f and g: we take $f(\vec{x}) = 2^{\sum_{i=0}^{n-1} x_i \cdot 2^i}$, and g such that $f(\vec{x}) \times g(\vec{x}) = 2^{2^{n+1}-1}$.

Both functions have $\mathtt{SLDD}_\times$ representations with $n+1$ nodes and $2n$ arcs, and only exponential AADD representations (following the same mechanism as in the previous proof). Since a $\mathtt{SLDD}_\times$ representation can be transformed into an AADD one in linear time (Proposition C.5), the results follow. □

Proposition C.31.

- $\mathtt{SLDD}_+$ *satisfies* $+\mathbf{BC}$*;*
- $\mathtt{SLDD}_\times$ *satisfies* $\times\mathbf{BC}$*;*
- ADD *satisfies* $\times\mathbf{BC}$*,*$+\mathbf{BC}$*,* $\max\mathbf{BC}$*,* $\min\mathbf{BC}$*.*

Proof. Let α and α' be two $\mathtt{SLDD}_\otimes$ formulæ, over $\{x_1, \ldots, x_n\}$, with $\otimes \in \{+, \times\}$. They are supposed to be ordered in the same way, using the ordering $x_1 \lhd \cdots \lhd x_n$. We aim at building an $\mathtt{SLDD}_\otimes$ representation of $g = f_\alpha^{\mathtt{SLDD}_\otimes} \otimes f_{\alpha'}^{\mathtt{SLDD}_\otimes}$ based on the same variable ordering.

First, we are going to modify α (resp. α') such that every path from the root to the leaf of the formula is of the form

$\langle a_1, \ldots, a_n \rangle$ where each a_i (with $i \in \{1, \ldots, n-1\}$) is an arc from a node labeled with x_i to a node labeled with x_{i+1}, and a_n is an arc from a node labeled with x_n to the leaf node. Lemma C.22 states that this can be done in polynomial time.

Consider now an assignment $\vec{x}$ and let $p(\vec{x}) = \langle a_1, \ldots, a_n \rangle$ and $p'(\vec{x}) = \langle a'_1, \ldots, a'_n \rangle$ be the corresponding paths in α and α'. It holds that:

$$g(\vec{x}) = (\varphi(a_1) \otimes \cdots \otimes \varphi(a_n)) \otimes (\varphi(a'_1) \otimes \cdots \otimes \varphi(a'_n)).$$

Because $\otimes$ is associative and commutative, we get:

$$g(\vec{x}) = (\varphi(a_1) \otimes \varphi(a'_1)) \otimes \cdots \otimes (\varphi(a_n) \otimes \varphi(a'_n)).$$

It is thus possible to get an $\texttt{SLDD}_\otimes$ representation of g by making the product of the two graphs, levelwise: for any N of α and N' of α' such that $\mathrm{Var}(N) = \mathrm{Var}(N') = x$, the new graph, δ, contains the node denoted $N \otimes N'$ such that $\mathrm{Var}(N \otimes N') = x$; we add a leaf labeled with the neutral element for $\otimes$. For each value d in the domain of x, let a be the arc in $\mathrm{Out}(N)$ such that $v(a) = d$ and a' be the arc in $\mathrm{Out}(N')$ such that $v(a') = d$; add to the new graph one arc a_δ from node $N \otimes N'$ to node $M \otimes M'$ with value $v(a_\delta) = d$ and $\varphi(a_\delta) = \varphi(a) \otimes \varphi(a')$; then recursively delete unreachable nodes and arcs. The offset of δ is set to $\mathrm{Offset}(\alpha) \otimes \mathrm{Offset}(\alpha')$. This construction is feasible in time polynomial in the sizes of α and α'.

The resulting structure δ is a (typically non-normalized) $\texttt{SLDD}_\otimes$ formula. For any assignment $\vec{x}$, consider the corresponding path $p(\vec{x}) = \langle a''_1, \ldots, a''_n \rangle$ in δ. We have:

$$\begin{aligned}
f_\delta^{\texttt{SLDD}_\otimes}(\vec{x}) &= \mathrm{Offset}(\delta) \otimes \varphi(a''_1) \otimes \cdots \otimes \varphi(a''_n) \\
&= \mathrm{Offset}(\alpha) \otimes \mathrm{Offset}(\alpha') \\
&\quad \otimes \varphi(a_1) \otimes \varphi(a'_1) \otimes \cdots \otimes \varphi(a_n) \otimes \varphi(a'_n) \\
&= (\mathrm{Offset}(\alpha) \otimes \varphi(a_1) \otimes \cdots \otimes \varphi(a_n)) \\
&\quad \otimes (\mathrm{Offset}(\alpha') \otimes \varphi(a'_1) \otimes \cdots \otimes \varphi(a'_n)) \\
&= f_\alpha^{\texttt{SLDD}_\otimes}(\vec{x}) \otimes f_{\alpha'}^{\texttt{SLDD}_\otimes}(\vec{x}) \\
&= (f_\alpha^{\texttt{SLDD}_\otimes} \otimes f_{\alpha'}^{\texttt{SLDD}_\otimes})(\vec{x}).
\end{aligned}$$

That is to say, δ is an $\texttt{SLDD}_+$ representation of $f_\alpha^{\texttt{SLDD}_\otimes} \otimes f_{\alpha'}^{\texttt{SLDD}_\otimes}$. This proves that $\texttt{SLDD}_+$ satisfies $+\mathbf{BC}$ and $\texttt{SLDD}_\times$ satisfies $\times\mathbf{BC}$.

Bounded combinations by $\otimes \in \{\min, \max, +, \times\}$ hold on ADD formulæ, using the same idea of automata product. There are no valuations on the arcs, but on the leaf nodes: for each leaf N in α and each leaf M in β, the leaves of δ are nodes $N \otimes M$ labeled with $\varphi(N \otimes M) = \varphi(N) \otimes \varphi(M)$. □

Sentential Entailment

Proposition C.32. ADD *satisfies* **SE**.

Proof. Let α and β be two ADD formulæ. We have to show that it is possible to decide in polynomial time whether $\forall \vec{x}, f_\alpha(\vec{x}) \leq f_\beta(\vec{x})$. The point is that this property holds if and only if for every value γ taken by f_α on some assignments $\vec{x}$, the set of these assignments is included in the set of assignments $\vec{y}$ such that $f_\beta(\vec{y}) \geq \gamma$. So, let Γ be the (finite) set of labels of the terminal nodes of α. For each level $\gamma \in \Gamma$, thanks to Propositions C.9 and C.6, we compute in linear time an MDD formula $\alpha_{=\gamma}$ representing $CUT^{\sim\gamma}(f_\alpha)$ (its 1-leaf is the leaf of α labeled with γ and its 0-leaf is obtained by merging all the leaves of α not labeled with γ) and an MDD formula $\beta_{\geq\gamma}$ representing $CUT^{\succeq\gamma}(f_\beta)$ (its 1-leaf is obtained by merging all the leaves of β labeled with a $\lambda \geq \gamma$ and its 0-leaf is obtained by merging all the leaves of α not labeled with a $\lambda < \gamma$). Checking whether every model of $\alpha_{=\gamma}$ is a model of $\beta_{\geq\gamma}$ can be done in polynomial time since **SE** is satisfied by MDD.

The procedure repeats this operation for each element of Γ, thus it runs in polynomial time. If it is the case that for each $\gamma \in \Gamma$, every model of $\alpha_{=\gamma}$ is a model of $\beta_{\geq\gamma}$, then it means that $\forall \vec{x}, f_\alpha(\vec{x}) \leq f_\beta(\vec{x})$, so the procedure outputs 1; if it is not the case, it outputs 0. □

Proposition C.33. $\texttt{SLDD}_\times$ *satisfies* **SE**.

Proof. First, note that $\forall \vec{x}, f(\vec{x}) \leq g(\vec{x})$ holds if and only if (i) $\forall \vec{x}, g(\vec{x}) = 0 \implies f(\vec{x}) = 0$, and (ii) $\forall \vec{x}, f \times g'(\vec{x}) \leq 1$, where $g'(\vec{x}) = 1/g(\vec{x})$ if $g(\vec{x}) > 0$ and 0 otherwise; this is not hard to check in both directions. Then, testing whether $\forall \vec{x}, f_\alpha(\vec{x}) \leq f_\beta(\vec{x})$ amounts to verify that both conditions hold.

(i) To verify that $\forall \vec{x}, f_\beta(\vec{x}) = 0 \implies f_\alpha(\vec{x}) = 0$, we only have to compute the minimal value taken by f_β (this is polynomial, since $\texttt{SLDD}_\times$ satisfies $\mathbf{OPT}_{\min}$, see Proposition C.11), and if it is 0, to compute MDD formulæ α' and β' representing the min-cuts of α and β, respectively (this is polynomial, since by Proposition C.12, $\texttt{SLDD}_\times$ satisfies $\mathbf{CUT}_{\min}$, and by Proposition C.6, an $\texttt{SLDD}_\times$ formula on $\{0, 1\}$ can be transformed into an equivalent MDD in polynomial time), and to check whether $\beta' \models \alpha'$, again in polynomial time as MDD satisfies **SE** (Amilhastre et al. 2014).

(ii) To verify that $\forall \vec{x}, f \times g'(\vec{x}) \leq 1$, we first compute an $\texttt{SLDD}_\times$ representation δ of g' by inverting the offset and the label of every arc in f_β (that is, V becomes $1/V$), except for 0-labels that remain unchanged (any path in δ corresponds to the value $\prod_i 1/\varphi(a_i) = 1/\prod_i \varphi(a_i)$, except when it contains a 0-arc, so $f_\delta = g'$). It is then easy to check whether $\forall \vec{x}, f_\alpha(\vec{x}) \times g'(\vec{x}) \leq 1$: just compute the $\times$-combination of α and δ (this is polynomial because $\texttt{SLDD}_\times$ satisfies $\times\mathbf{BC}$, see Proposition C.31), compute the maximal value v^* taken by the resulting formula (polynomial because $\texttt{SLDD}_\times$ satisfies $\mathbf{OPT}_{\max}$, see Proposition C.11) and check whether $v^* \leq 1$. □

Proposition C.34. $\texttt{SLDD}_+$ *satisfies* **SE**.

Proof. First, note that $\forall \vec{x}, f(\vec{x}) \leq g(\vec{x})$ holds if and only if $\forall \vec{x}, K \leq g(\vec{x}) + K - f(\vec{x})$, where K is some constant. Thus, in order to check whether $\forall \vec{x}, f_\alpha(\vec{x}) \leq f_\beta(\vec{x})$, we are going to build an $\texttt{SLDD}_+$ formula α' such that $f_{\alpha'} = K - f_\alpha$, with K large enough for $f_{\alpha'}$ to range over $\mathbb{R}^+$. We compute v^*, the maximal value taken by f_α (this can be done in polynomial time, since $\texttt{SLDD}_+$ satisfies $\mathbf{OPT}_{\max}$,

see Proposition C.11), and then choose some $K \geq v^*$. Applying Lemma C.23, we get an $\mathtt{SLDD}_+$ representation α' of the function $K - f_\alpha$. We then only have to build the $+$-combination of α' and β (polynomial since $\mathtt{SLDD}_+$ satisfies $+\mathbf{BC}$, see Proposition C.31) and check that the minimal value taken by the resulting formula is larger than K (polynomial since $\mathtt{SLDD}_+$ satisfies $\mathbf{OPT}_{\min}$, see Proposition C.11). □

Variable Elimination

Lemma C.35. *For each language* L *among* $\{\mathtt{ADD}, \mathtt{SLDD}_+, \mathtt{SLDD}_+, \mathtt{AADD}\}$*, and each operator* $\odot \in \{\max, \min, +, \times\}$*,*

- L *satisfies* $\mathbf{S}\odot\mathbf{Elim}$ *if and only if it satisfies* $\odot\mathbf{C}$*;*
- L *satisfies* $\mathbf{SB}\odot\mathbf{Elim}$ *if and only if it satisfies* $\odot\mathbf{BC}$*;*

Proof. We first prove the two sufficient conditions, then the two necessary conditions.

(⇒) If the $\odot$-elimination of a single variable in an L formula Σ can be achieved in polynomial time, it is possible to compute in polynomial time the $\odot$-combination of any set $\{\alpha_1, \ldots, \alpha_n\}$ of L formulæ. Indeed, from $\{\alpha_1, \ldots, \alpha_n\}$, we can generate in linear time the following L formula Σ: its root N_0 is labeled with a (new) variable v not occurring in any α_i, with $D_v = \{1, \ldots, n\}$; N_0 has n outgoing arcs a_i (with $i \in \{1, \ldots, n\}$)), where each a_i corresponds to $v = i$ and points to the root of α_i, and the leaves of $\alpha_1, \ldots, \alpha_n$ are merged when they have the same label; finally, when $\mathtt{L} = \mathtt{AADD}$ (resp. $\mathtt{SLDD}_+$, resp. $\mathtt{SLDD}_\times$), the label of each a_i is set to $\langle 0, 1\rangle$ (resp. 0, resp. 1). By definition, the $\odot$-elimination of v in Σ is $f^{\mathtt{L}}_{\Sigma,\langle v,1\rangle} \odot \cdots \odot f^{\mathtt{L}}_{\Sigma,\langle v,n\rangle}$, which by construction is equal to $f^{\mathtt{L}}_{\alpha_1} \odot \cdots \odot f^{\mathtt{L}}_{\alpha_n}$. Accordingly, if we can obtain in polynomial time an L representation of the $\odot$-elimination of v in Σ, we can also obtain in polynomial time an L representation of the $\odot$-combination of $\{\alpha_1, \ldots, \alpha_n\}$.

This proves that the satisfaction of $\mathbf{S}\odot\mathbf{Elim}$ implies that of $\odot\mathbf{C}$; now, note that the exact same construction also works in the bounded case. Indeed, when $n = 2$, the $\odot$-elimination of v in Σ, which can be obtain in time polynomial in $|\Sigma|^2$ if L satisfies $\mathbf{SB}\odot\mathbf{Elim}$, is $f^{\mathtt{L}}_{\Sigma,\langle v,1\rangle} \odot f^{\mathtt{L}}_{\Sigma,\langle v,2\rangle}$, which by construction is equal to $f^{\mathtt{L}}_{\alpha_1} \odot f^{\mathtt{L}}_{\alpha_2}$. This proves that the satisfaction of $\mathbf{SB}\odot\mathbf{Elim}$ implies that of $\odot\mathbf{BC}$.

(⇐) Consider an L formula α and a variable $x \in \mathrm{Var}(\alpha)$ with a domain of size d. Since L satisfies $\mathbf{CD}$ (Proposition C.8),[3] we can build in time polynomial an L representation of $f^{\mathtt{L}}_{\alpha,\vec{x}}$ for each of the d possible $\vec{x}$; we denote them as $\alpha_1, \ldots, \alpha_d$.

Now, if L satisfies $\odot\mathbf{C}$, we can obtain an L representation β of $\bigodot_{\vec{x}} f^{\mathtt{L}}_{\alpha,\vec{x}}$ in time polynomial in $\sum_{i=1}^{d} |\alpha_i|$, which is polynomial in $|\alpha|$ (each α_i is of size polynomial in $|\alpha|$, and $d \leq |\alpha|$ since there is at least one x-node in α, that has by definition d outgoing arcs). By definition, β is an L representation of the $\odot$-elimination of x in α, so L satisfies $\odot\mathbf{C}$.

[3] Note that conditioning can actually be achieved in linear time, without ever increasing the size of the formula. But the proof would work even if it were not the case.

If L only satisfies $\odot\mathbf{BC}$, then we can also build β, but we have to do it incrementally with $d - 1$ binary $\odot$-combinations, so the bound is looser: $\alpha_1 \odot \cdots \odot \alpha_d$ is obtained in time bounded by a polynomial of $|\alpha|^d$. However, this is enough for L to satisfy $\mathbf{SB}\odot\mathbf{Elim}$, by definition. □

Proposition C.36. $\mathtt{ADD}$*,* $\mathtt{SLDD}_+$*,* $\mathtt{SLDD}_\times$*, and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{S}\odot\mathbf{Elim}$ *or* $\odot\mathbf{Elim}$ *for any* $\odot \in \{\max, \min, +, \times\}$*.*

Proof. For any $\odot \in \{\max, \min, +, \times\}$, the satisfaction of $\odot\mathbf{Elim}$ implies that of $\mathbf{S}\odot\mathbf{Elim}$, and the satisfaction of $\mathbf{S}\odot\mathbf{Elim}$ implies that of $\odot\mathbf{C}$ (Lemma C.35); yet, we have shown that $\mathtt{ADD}$, $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, and $\mathtt{AADD}$ do not satisfy $\odot\mathbf{C}$ when $\odot \in \{\max, \min, +, \times\}$ (Proposition C.28). □

Proposition C.37. *The following results hold:*

- $\mathtt{ADD}$ *satisfies* $\mathbf{SB}\odot\mathbf{Elim}$ *for* $\odot \in \{\times, +, \min, \max\}$*;*
- $\mathtt{SLDD}_+$ *satisfies* $\mathbf{SB}{+}\mathbf{Elim}$*;*
- $\mathtt{SLDD}_+$ *and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{SB}{\times}\mathbf{Elim}$*;*
- $\mathtt{SLDD}_\times$ *satisfies* $\mathbf{SB}{\times}\mathbf{Elim}$*;*
- $\mathtt{SLDD}_\times$ *and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{SB}{+}\mathbf{Elim}$*;*
- $\mathtt{SLDD}_+$*,* $\mathtt{SLDD}_\times$*, and* $\mathtt{AADD}$ *do not satisfy* $\mathbf{SB}\odot\mathbf{Elim}$ *for* $\odot \in \{\max, \min\}$*.*

Proof. These are direct consequences of Lemma C.35, since

- $\mathtt{ADD}$ satisfies $\odot\mathbf{BC}$ for $\odot \in \{\times, +, \min, \max\}$ (Proposition C.31);
- $\mathtt{SLDD}_+$ satisfies $+\mathbf{BC}$ (Proposition C.31);
- $\mathtt{SLDD}_+$ and $\mathtt{AADD}$ do not satisfy $\times\mathbf{BC}$ (Proposition C.30);
- $\mathtt{SLDD}_\times$ satisfies $\times\mathbf{BC}$ (Proposition C.31);
- $\mathtt{SLDD}_\times$ and $\mathtt{AADD}$ do not satisfy $+\mathbf{BC}$ (Proposition C.30);
- $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, and $\mathtt{AADD}$ do not satisfy $\max\mathbf{BC}$ or $\min\mathbf{BC}$ (Proposition C.29). □

Marginalization

The marginalization proofs use the fact that the $\odot$-elimination of every variable in any L formula α (i.e., the "full" variable elimination) can be done in polynomial time. We denote as $\mathrm{Elim}_\odot(\alpha)$ the value resulting from such a "full" variable elimination, that is, the value $\bigodot_{\vec{x} \in D_{\mathrm{Var}_{\mathtt{L}}(\alpha)}} f^{\mathtt{L}}_{\alpha,\vec{x}}$.

Lemma C.38. *For any language* $\mathtt{L} \in \{\mathtt{ADD}, \mathtt{SLDD}_+, \mathtt{SLDD}_\times, \mathtt{AADD}\}$ *and any operator* $\odot \in \{\max, \min, +, \times\}$*, if there exists a polynomial algorithm mapping any* L *formula* α *to the value* $\mathrm{Elim}_\odot(\alpha)$*, then* L *satisfies* $\odot\mathbf{Marg}$*.*

Proof. Consider an L formula α and a variable x, denoting denoting $D_x = \{d_1, \ldots, d_k\}$. The following $\mathtt{ADD}$ formula β is a representation of the $\odot$-marginalization of $f^{\mathtt{L}}_\alpha$ on x: β has one root labeled with x, with k outgoing arcs $a_1, \ldots, a_k$, such that for each $i \in \{1, \ldots, k\}$, $v(a_i) = d_i$ and a_i points to a leaf L_i with value $\varphi(L_i) = \bigodot_{\vec{y} \in D_{\mathrm{Var}(\alpha)\setminus\{x\}}} f^{\mathtt{L}}_{\alpha,\vec{y}}(\langle x, d_i\rangle)$, that is, $\varphi(L_i) = \mathrm{Elim}_\odot(f^{\mathtt{L}}_{\alpha,\langle x,d_i\rangle})$. Accordingly, if for any value d_i, the valuation $\varphi(L_i)$ can be computed in time polynomial in the size of α (i.e., if "full" variable elimination can be computed in

polynomial time from α once conditioned by $x = d_i$), then β can be computed in time polynomial in the size of α.

Since all four languages satisfy $\mathbf{CD}$ (Proposition C.8), this implies that if there exists a polynomial algorithm for "full" variable elimination, then there exists a polynomial algorithm building an ADD representation β of the marginalization of any L formula on x. This shows the result when L is ADD; now when L is $\mathtt{SLDD}_+$ (resp. $\mathtt{SLDD}_\times$, resp. AADD), β can be turned in linear time into an equivalent $\mathtt{SLDD}_+$ (resp. $\mathtt{SLDD}_\times$, resp. AADD) formula (Proposition C.5). □

Proposition C.39. ADD, $\mathtt{SLDD}_+$, $\mathtt{SLDD}_\times$, *and* AADD *satisfy* $\odot\mathbf{Marg}$ *for* $\odot \in \{\max, \min\}$.

Proof. When $\odot$ is $\max$ (resp. $\min$), $\mathrm{Elim}_\odot(\alpha)$ is simply the maximal (resp. minimal) value taken by $f_\alpha^{\mathtt{L}}$. Since from Proposition C.11, each language in $\{\mathtt{ADD}, \mathtt{SLDD}_+, \mathtt{SLDD}_\times, \mathtt{AADD}\}$ satisfies $\mathbf{OPT}_{\max}$ (resp. $\mathbf{OPT}_{\min}$), "full" variable $\max$-elimination (resp. $\min$-elimination) can be done in polynomial time; hence from Lemma C.38 we get that these languages all satisfy $\max\mathbf{Marg}$ (resp. $\min\mathbf{Marg}$). □

Proposition C.40. *There exists a polynomial-time algorithm mapping any* AADD *formula* α *to* $\mathrm{Elim}_+(\alpha)$.

Proof. We show that "full" variable $+$-elimination on AADD is polynomial because we can iteratively eliminate the last variable in linear time.

Let $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{X}$ and $y \in \mathcal{X}$; let α be an AADD formula over $X \cup \{y\}$, ordered in such a way that $x_1 \lhd \cdots \lhd x_n \lhd y$. To simplify the proof, we suppose that every variable in $X \cup \{y\}$ is mentioned in every path of α; this is harmless, since we never need to add more than $(n+1)d$ arcs (with φ-value $\langle 0, 1\rangle$) per node in α (with d the cardinal of the largest variable domain).

We have the following:

$$\begin{aligned}\mathrm{Elim}_+(\alpha) &= \sum_{\vec{z} \in D_X \times D_y} f_\alpha^{\mathtt{AADD}}(\vec{z}) \\ &= \sum_{\vec{x} \in D_X} \sum_{\vec{y} \in D_y} f_\alpha^{\mathtt{AADD}}(\vec{x} \cdot \vec{y})\end{aligned}$$

For a given assignment $\vec{x} \cdot \vec{y}$, let us consider the path p in α corresponding to $\vec{x} \cdot \vec{y}$. The path p contains $n+1$ arcs, that we denote $a_1, \ldots, a_n, a_{n+1}$. For each $i \in \{1, \ldots, n+1\}$, we denote $\varphi(a_i) = \langle q_i, f_i\rangle$; finally, the offset is as usual $\langle q_0, f_0\rangle$. By definition of the interpretation function of AADD,

$$\begin{aligned}f_\alpha^{\mathtt{AADD}}(\vec{x} \cdot \vec{y}) &= q_0 + f_0(q_1 + f_1(q_2 + \cdots \\ &\qquad \cdots + f_n(q_{n+1} + f_{n+1} \times 0) \cdots)) \\ &= q_0 + f_0 q_1 + f_0 f_1 q_2 + \cdots \\ &\qquad \cdots + f_0 \cdots f_n q_{n+1} \\ &= \sum_{i=0}^{n+1} \left(q_i \prod_{j=0}^{i-1} f_j \right) \\ &= \sum_{i=0}^{n} \left(q_i \prod_{j=0}^{i-1} f_j \right) + q_{n+1} \cdot \prod_{j=0}^{n} f_j.\end{aligned}$$

There are three elements in this formula:

- the valuation q_{n+1} of the y-arc, which is the last arc along the path, and which we denote as $\varphi_{\vec{x} \cdot \vec{y}}$;
- the "additive offset" given by the first n arcs, $\sum_{i=0}^{n} \left(q_i \prod_{j=0}^{i-1} f_j \right)$, which we denote as $Q_{\vec{x}}$;
- the "multiplicative offset" given by the first n arcs, $\prod_{j=0}^{n} f_j$, that we denote as $F_{\vec{x}}$.

The key point is that neither $Q_{\vec{x}}$ nor $F_{\vec{x}}$ depends on $\vec{y}$, so we can write

$$\begin{aligned}\mathrm{Elim}_+(\alpha) &= \sum_{\vec{x} \in D_X} \sum_{\vec{y} \in D_y} f_\alpha^{\mathtt{AADD}}(\vec{x} \cdot \vec{y}) \\ &= \sum_{\vec{x} \in D_X} \sum_{\vec{y} \in D_y} (Q_{\vec{x}} + \varphi_{\vec{x} \cdot \vec{y}} \cdot F_{\vec{x}}) \\ &= \sum_{\vec{x} \in D_X} \left(|D_y| \cdot Q_{\vec{x}} + F_{\vec{x}} \cdot \sum_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}} \right). \qquad (1)\end{aligned}$$

It is thus possible to transform α into another AADD formula α' that does not mention y and verifies $\mathrm{Elim}_+(\alpha') = \mathrm{Elim}_+(\alpha)$ as follows. For each y-labeled node N, we compute the value $\varphi_N = \sum_{a \in \mathrm{Out}(N)} q_a$. Then we update the φ-label of each arc $a_{\mathrm{In}} \in \mathrm{In}(N)$ so that it becomes $\langle q_{a_{\mathrm{In}}} + \frac{f_{a_{\mathrm{In}}}}{|D_y|} \varphi_N, 0\rangle$. We also modify the offset, which becomes $\langle q_0 \cdot |D_y|, f_0 \cdot |D_y|\rangle$. Finally, we merge all the y-nodes into a new leaf.

We show that $\mathrm{Elim}_+(\alpha') = \mathrm{Elim}_+(\alpha)$; let us denote $\langle q', f'\rangle$ the φ-label of an arc in α' when the corresponding arc in α has label $\langle q, f\rangle$. We can rewrite the left part of the sum in Equation 1:

$$\begin{aligned}|D_y| \cdot Q_{\vec{x}} &= |D_y| \cdot \sum_{i=0}^{n} \left(q_i \prod_{j=0}^{i-1} f_j \right) \\ &= |D_y| \cdot q_0 + \sum_{i=1}^{n} \left(q_i \cdot |D_y| \cdot f_0 \prod_{j=1}^{i-1} f_j \right) \\ &= q'_0 + \sum_{i=1}^{n} \left(q_i f'_0 \prod_{j=1}^{i-1} f_j \right) \\ &= q'_0 + \sum_{i=1}^{n} \left(q_i \prod_{j=0}^{i-1} f'_j \right),\end{aligned}$$

since for $i \in \{1, \ldots, n-1\}$, $f'_i = f_i$. Isolating q_n, we get

$$\begin{aligned}|D_y| \cdot Q_{\vec{x}} &= q'_0 + \sum_{i=1}^{n-1} \left(q_i \prod_{j=0}^{i-1} f'_j \right) + q_n \prod_{j=0}^{n-1} f'_j \\ &= \sum_{i=0}^{n-1} \left(q'_i \prod_{j=0}^{i-1} f'_j \right) + q_n \prod_{j=0}^{n-1} f'_j,\end{aligned}$$

since for $i \in \{1, \dots, n-1\}$, $q'_i = q_i$. As for the right part of the sum in Equation 1:

$$F_{\vec{x}} \cdot \sum_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}} = f_0 \left(\prod_{j=1}^{n-1} f_j \right) f_n \sum_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}}$$
$$= \left(\prod_{j=0}^{n-1} f'_j \right) \frac{f_n}{|D_y|} \sum_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}},$$

since $f'_0 = f_0 \cdot |D_y|$ and for $i \in \{1, \dots, n-1\}$, $f'_i = f_i$. Now, since $q'_n = q_n + \frac{f_n}{|D_y|} \sum_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}}$, by summing the two parts, we get

$$|D_y| \cdot Q_{\vec{x}} + F_{\vec{x}} \cdot \sum_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}}$$
$$= \sum_{i=0}^{n-1} \left(q'_i \prod_{j=0}^{i-1} f'_j \right) + \left(\prod_{j=0}^{n-1} f'_j \right) q'_n$$
$$= \sum_{i=0}^{n} \left(q'_i \prod_{j=0}^{i-1} f'_j \right) = f_{\alpha'}^{\texttt{AADD}}(\vec{x}),$$

by definition of the interpretation function of AADD. Equation 1 can hence be rewritten as

$$\mathrm{Elim}_+(\alpha) = \sum_{\vec{x} \in D_X} f_{\alpha'}^{\texttt{AADD}}(\vec{x}) = \mathrm{Elim}_+(\alpha').$$

The procedure thus eliminates the last variable without changing the value of the "full" variable elimination; moreover, it runs in linear time, and the resulting formula is always smaller than the original one. Hence, iteratively repeating the procedure for each variable in a bottom-up way, and stopping when the resulting formula only contains a leaf and an offset, we obtain $\mathrm{Elim}_+(\alpha)$ in time polynomial in the size of α. □

Corollary C.41. ADD, $\texttt{SLDD}_+$, $\texttt{SLDD}_\times$, *and* AADD *satisfy* $+\mathbf{Marg}$.

Proof. Full variable $+$-elimination is polynomial on AADD. Since any ADD (resp. $\texttt{SLDD}_+$, $\texttt{SLDD}_\times$) formula can be turned into an equivalent AADD formula in linear time (Proposition C.5), full variable $+$-elimination is also polynomial on ADD (resp. $\texttt{SLDD}_+$, $\texttt{SLDD}_\times$). Hence, from Lemma C.38, we get that all four languages satisfy $+\mathbf{Marg}$. □

Proposition C.42. *There exists a polynomial-time algorithm mapping any* $\texttt{SLDD}_\times$ *formula* α *to* $\mathrm{Elim}_\times(\alpha)$.

Proof. The proof is similar to that of full $+$-elimination in AADD: we show that we can iteratively eliminate the last variable in linear time. Let $X = \{x_1, \dots, x_n\} \subseteq \mathcal{X}$ and $y \in \mathcal{X}$; let α be an $\texttt{SLDD}_\times$ formula over $X \cup \{y\}$, ordered in such a way that $x_1 \lhd \cdots \lhd x_n \lhd y$. To simplify the proof, we suppose that every variable in $X \cup \{y\}$ is mentioned in every path of α. Lemma C.22 shows that this can be done in polynomial time.

We have the following:

$$\mathrm{Elim}_\times(\alpha) = \prod_{\vec{z} \in D_X \times D_y} f_\alpha^{\texttt{SLDD}_\times}(\vec{z})$$
$$= \prod_{\vec{x} \in D_X} \prod_{\vec{y} \in D_y} f_\alpha^{\texttt{SLDD}_\times}(\vec{x} \cdot \vec{y})$$

For a given assignment $\vec{x} \cdot \vec{y}$, let us consider the path p in α corresponding to $\vec{x} \cdot \vec{y}$. The path p contains $n+1$ arcs: we denote as $\Phi_{\vec{x}}$ the product of the φ-labels of the first n arcs and the offset, and as $\varphi_{\vec{x} \cdot \vec{y}}$ the φ-label of the last arc. By definition of the interpretation function of $\texttt{SLDD}_\times$,

$$f_\alpha^{\texttt{SLDD}_\times}(\vec{x} \cdot \vec{y}) = \Phi_{\vec{x}} \times \varphi_{\vec{x} \cdot \vec{y}},$$

and since $\Phi_{\vec{x}}$ does not depend on $\vec{y}$, we get

$$\mathrm{Elim}_\times(\alpha) = \prod_{\vec{x} \in D_X} \prod_{\vec{y} \in D_y} \Phi_{\vec{x}} \times \varphi_{\vec{x} \cdot \vec{y}}$$
$$= \prod_{\vec{x} \in D_X} \left(\Phi_{\vec{x}} \prod_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}} \right).$$

Hence, it is possible to transform α into another $\texttt{SLDD}_\times$ formula α' that does not mention y and verifies $\mathrm{Elim}_\times(\alpha') = \mathrm{Elim}_\times(\alpha)$ as follows. For each y-labeled node N, we compute the value $\varphi_N = \times_{a \in \mathrm{Out}(N)} q_a$, then we update the φ-label of each arc $a_{\mathrm{In}} \in \mathrm{In}(N)$ so that it becomes $\varphi(a_{\mathrm{In}}) \times \varphi_N$, and finally, we merge all the y-nodes into a new leaf. The process is linear in the size of α, and it should be clear that

$$f_{\alpha'}^{\texttt{SLDD}_\times}(\vec{x}) = \Phi_{\vec{x}} \prod_{\vec{y} \in D_y} \varphi_{\vec{x} \cdot \vec{y}},$$

and thus that

$$\mathrm{Elim}_\times(\alpha) = \prod_{\vec{x} \in D_X} f_{\alpha'}^{\texttt{SLDD}_\times}(\vec{x}) = \mathrm{Elim}_\times(\alpha').$$

Consequently, it is possible to eliminate the last variable in an $\texttt{SLDD}_\times$ formula in linear time, without changing the value of the "full" variable elimination. Computing this value can thus be done by eliminating every variable iteratively in a bottom-up way, stopping when the formula is reduced to a leaf and an offset; since the size of the formula is always decreasing, the overall procedure is polynomial-time. □

Corollary C.43. ADD *and* $\texttt{SLDD}_\times$ *satisfy* $\times\mathbf{Marg}$.

Proof. Full variable $\times$-elimination is polynomial on $\texttt{SLDD}_\times$. Since any ADD formula can be turned into an equivalent $\texttt{SLDD}_\times$ formula in linear time (Proposition C.5), full variable $\times$-elimination is also polynomial on ADD. Hence, from Lemma C.38, we get that ADD and $\texttt{SLDD}_\times$ satisfy $+\mathbf{Marg}$. □

Compilation de préférences
application à la configuration de produit

L'intérêt des différents langages de la famille des diagrammes de décision valués (VDD) est qu'ils admettent des algorithmes en temps polynomial pour des traitements (comme l'optimisation, la cohérence inverse globale, l'inférence) qui ne sont pas polynomiaux (sous l'hypothèse $P \neq NP$), si ils sont effectués sur le problème dans sa forme originale tel que les réseaux de contraintes ou les réseaux bayésiens.

Dans cette thèse, nous nous intéressons au problème de configuration de produit, et plus spécifiquement, la configuration en ligne avec fonction de valuation associée (typiquement, un prix). Ici, la présence d'un utilisateur en ligne nous impose une réponse rapide à ses requêtes, rapidité rendant impossible l'utilisation de langages n'admettant pas d'algorithmes en temps polynomial pour ces requêtes. La solution proposée est de compiler hors-ligne ces problèmes vers des langages satisfaisant ces requêtes, afin de diminuer le temps de réponse pour l'utilisateur.

Une première partie de cette thèse est consacrée à l'étude théorique des VDD, et plus particulièrement les trois langages *Algebraic Decision Diagrams*, *Semi ring Labelled Decision Diagrams* et *Affine Algebraic Decision Diagrams* (ADD, SLDD et AADD). Nous y remanions le cadre de définition des SLDD, proposons des procédures de traductions entre ces langages, et étudions la compacité théorique de ces langages. Nous établissons dans une deuxième partie la carte de compilation de ces langages, dans laquelle nous déterminons la complexité algorithmique d'un ensemble de requêtes et transformations correspondant à nos besoins. Nous proposons également un algorithme de compilation à approche ascendante, ainsi que plusieurs heuristiques d'ordonnancement de variables et contraintes visant à minimiser la taille de la représentation après compilation ainsi que le temps de compilation. Enfin la dernière partie est consacrée à l'étude expérimentale de la compilation et de l'utilisation de formes compilées pour la configuration de produit. Ces expérimentations confirment l'intérêt de notre approche pour la configuration en ligne de produit.

Nous avons implémenté au cours de cette thèse un compilateur (le compilateur SALADD) pleinement fonctionnel, réalisant la compilation de réseaux de contraintes et de réseaux bayésiens, et avons développés un ensemble de fonctions adaptées à la configuration de produit. Le bon fonctionnement et les bonnes performances de ce compilateur ont été validés via un protocole de validation commun à plusieurs solveurs.

Mots-clefs : Compilation, configuration de produit, recommandation, diagramme de décision valué, heuristique d'ordonnancement, CSP