# Maintenir des MDD persistants pour établir la consistance d'arc

**Julien Vion, Sylvain Piechowiak**

*Université de Valenciennes et du Hainaut Cambrésis, LAMIH CNRS UMR 8201*
*{julien.vion,sylvain.piechowiak}@univ-valenciennes.fr*

RÉSUMÉ. *Dans cet article, nous présentons* MDDF*, un nouvel algorithme de révision de contraintes définies sous forme de diagramme de décision multi-valué (MDD). En réalisant des copies partielles à la volée des MDD modifiés au cours de la recherche,* MDDF *parvient à une meilleure incrémentalité que* MDDC *proposé par Cheng et Yap (2010). La meilleure incrémentalité, couplée à la nouvelle possibilité d'exploiter les informations sur les variables modifiées permet d'éviter de parcourir systématiquement toute la structure de données à chaque filtrage, et ce malgré l'utilisation d'une file de propagation à gros grain. Outre ses performances sur les contraintes définies sous forme de MDD, les performances de* MDDF *sont proches de* STR2 *de Ullmann (2007) et Lecoutre (2011) sur les contraintes* table *peu dures et non structurées.*

ABSTRACT. *In this paper, we present* MDDF*, a new algorithm for revising constraints defined using multi-valued decision diagrams (MDD).* MDDF *copies modified parts of the MDD "on the fly" during the search of a solution, which yields a better incrementality than the previous* MDDC *algorithm by Cheng et Yap (2010), although coarse-grained propagation queues are still sufficient.* MDDF *can also be used to compress constraints defined as tables of allowed tuples.* MDDF *has then comparable performances to the* STR2 *algorithm by Ullmann (2007) and Lecoutre (2011) on loose, unstructured constraints.*

MOTS-CLÉS : *MDD, Propagation, CSP, Tables*

KEYWORDS: *MDD, Propagation, CSP, Tables*

DOI:10.3166/RIA.x.1-23 

## 1. Introduction

Le problème de satisfaction de contraintes (CSP pour *Constraint Satisfaction Problem*) est l'un des problèmes NP-complets de référence, à la fois pour sa clarté théorique et sa souplesse pour la modélisation des problèmes de décision combinatoires. Ceux-ci sont décomposés en variables et contraintes et l'objectif est d'affecter chaque variable à une valeur prise dans son domaine de sorte que toutes les contraintes soient satisfaites. La résolution des CSP discrets se fait généralement par exploration arborescente systématique et filtrage par consistance d'arc (AC pour *Arc Consistency*) de

chaque nœud. Rendre un CSP consistant consiste à supprimer toutes les valeurs qui ne peuvent apparaitre dant aucune solution. La consistance d'arc est une propriété qui considère une contrainte à la fois, c'est-à-dire un arc à la fois sous l'hypothèse historique d'un graphe de contraintes binaires. Les algorithmes de consistances d'arc se basent sur une file de propagation permettant de savoir quelles contraintes restent à réviser. Les files de propagation à gros grain maintiennent la liste des variables modifiées et/ou des contraintes à réviser. Les files de propagation à grain fin maintiennent la liste des valeurs supprimées entre deux appels à une procédure de révision (Boussemart, Hemery & Lecoutre, 2004). Dans cet article, nous restreignons le champ notre étude aux algorithmes à gros grain.

On généralise facilement le principe de la consistance d'arc aux réseaux non-binaires, mais dans ce cas le filtrage par consistance d'arc devient NP-difficile dans le cas général. Il existe cependant de nombreux cas particuliers de contraintes qu'il est possible de filtrer en temps et en espace polynomiaux. La contrainte *table* bénéficie d'un intérêt particulier : il s'agit d'une contrainte définie par une liste exhaustive de $k$-uplets qui vont satisfaire, ou au contraire invalider la contrainte, dite alors respectivement *positive* ou *négative*. On parle aussi de contrainte définie *en extension*. Les contraintes binaires qui étaient uniquement utilisées jusqu'au milieu des années 1990 étaient de ce type. L'algorithme de propagation générique AC-5 (van Hentenryck, Deville & Teng, 1992) a popularisé l'utilisation de contraintes plus spécialisées dites « globales », mais la contrainte *table* garde un grand intérêt théorique et elle trouve encore sa place en pratique dans la modélisation, par exemple, de problèmes de configuration ou de bases de données. De plus, certaines « tables » ayant une structure particulière peuvent être efficacement compressées et elles deviennent alors un moyen rapide, efficace et générique pour implanter des contraintes spécifiques, comme la contrainte *sequence*. La compression de tables sous la forme de diagrammes de décision multi-valués (MDD pour *Multi-valued Decision Diagrams*) est particulièrement prometteur (Cheng & Yap, 2010).

EXEMPLE 1. — La contrainte $C$, portant sur les variables $X$, $Y$ et $Z$, autorise six affectations de ces trois variables :

$$\begin{array}{lll} \mathrm{tab}(C) = \{ & \langle X=b, Y=c, Z=b\rangle\,, & \langle X=a, Y=b, Z=a\rangle\,, \\ & \langle X=a, Y=a, Z=a\rangle\,, & \langle X=c, Y=a, Z=c\rangle\,, \\ & \langle X=c, Y=a, Z=a\rangle\,, & \langle X=a, Y=a, Z=c\rangle\} \end{array}$$

□

Pour établir la consistance d'arc sur une telle contrainte, il faut montrer que chaque valeur de chaque variable dispose d'un *support*, c'est-à-dire d'un $k$-uplet *autorisé* et *valide*, où la valeur apparait. Un $k$-uplet autorisé est simplement un $k$-uplet apparaissant dans la table. Un $k$-uplet valide est un $k$-uplet dont toutes les valeurs apparaissent dans le domaine courant des variables concernées par la contrainte. Dans l'exemple 1, si le domaine de chaque variable est $\{a, b, c\}$, la valeur $b$ du domaine de $X$ est supportée par le premier $k$-uplet $\langle X = b, Y = c, Z = b\rangle$. Si $c$ est supprimée du domaine

de $Y$, ce $k$-uplet n'est plus valide, et la valeur $X = b$ n'est plus supportée : elle peut alors être filtrée, c'est-à-dire otée du domaine de $X$.

Lecoutre et Szymanek (2006) mettent en avant les deux stratégies principales pour trouver un support de chaque valeur : soit (a) on itère sur les $k$-uplets *valides*, en vérifiant à chaque itération si le $k$-uplet est autorisé par la contrainte ; soit (b) on itère sur les $k$-uplets *autorisés* par la contrainte, en vérifiant à chaque itération que le $k$-uplet est valide. Pour une contrainte portant sur $k$ variables dont les domaines n'ont pas plus de $d$ valeurs, et autorisant $\lambda$ $k$-uplets, l'approche (a) aura une complexité temporelle dans le pire des cas en $O(k(d^k - \lambda))$ ; l'approche (b) sera en $O(k\lambda)$. Il parait alors évident que l'approche (a) ne sera raisonnablement exploitable que lorsque la valeur de $\lambda$ est très grande, c'est-à-dire proche de $d^k$ (ce qui est généralement le cas pour des contraintes « négatives ») et l'approche (b) sera plus efficace pour les valeurs de $\lambda$ faibles, donc pour les contraintes « positives ».

Les algorithmes de consistance d'arc « standard » comme GAC-Schema (Bessière & Régin, 1997) ou AC-3$^{\text{rm}}$ (Lecoutre & Hemery, 2007) sont très satisfaisants pour implémenter l'approche (a), pour peu que l'on dispose d'une structure de données efficaces pour représenter la liste de $k$-uplets. On peut utiliser une simple table de hachage, mais des techniques plus efficaces existent (Lhomme & Régin, 2005 ; Katsirelos & Walsh, 2007). L'approche (b) a été plus particulièrement étudiée récemment : citons des algorithmes spécifiques comme *STR* (Ullmann, 2007), STR2 (Lecoutre, 2011), STR3 (Lecoutre, Likitvivatanavong & Yap, 2012), *AC5TC* (Mairy, van Hentenryck & Deville, 2014), *MDDC* (Cheng & Yap, 2010) ou *MDDI* (Gange, Stuckey & Szymanek, 2011). Il est également possible d'utiliser GAC-Schema ou AC-3$^{\text{rm}}$ avec l'approche (b) en implémentant une fonction de *recherche de support* dans une structure de données spécifique. C'est la stratégie adoptée par Gent, Jefferson, Miguel et Nightingale (2007), qui combine GAC-Schema avec la recherche de supports dans des Tries (cf section 5).

La représentation des tables par un diagramme de décision multi-valué (MDD) est l'une des approches les plus prometteuses : les MDD peuvent représenter un nombre exponentiel de $k$-uplets en espace polynomial, ce qui les rend très expressifs et rapides à traiter (Cheng & Yap, 2010). Dans cet article, nous nous intéressons plus particulièrement à l'approche (b) et en particulier aux algorithmes *STR2* et *MDDC*. Tous deux sont reconnus comme efficaces et surtout relativement simples à implanter et analyser, dans la mesure où ils ne nécessitent pas l'utilisation de files de propagation à grain fin (cf. section 3). *STR2* et *MDDC* se basent sur l'approche (b), tout en cherchant à « marquer », au cours de la recherche, les parties de leurs structures de données qu'il est inutile d'explorer. Ainsi, le maintien de la consistance d'arc au cours de la recherche devient partiellement incrémental, ce qui améliore considérablement les performances globales de l'algorithme. Après une description de ces deux algorithmes, nous proposons *MDDF*, qui combine les principes de *STR2* et *MDDC* pour établir la consistance d'arc sur une contrainte définie à partir d'un MDD.

## 2. Définitions et notations

DÉFINITION 2 (Réseau de contraintes, variable, domaine, contrainte). — *Un* réseau de contraintes *est un couple* $(\mathscr{X}, \mathscr{C})$ *dans lequel* $\mathscr{X}$ *est un ensemble de* $\boldsymbol{n}$ variables *; chaque variable* $X \in \mathscr{X}$ *est définie sur un domaine* $\mathrm{dom}(X)$ *;* $\mathscr{C}$ *est un ensemble de* $\boldsymbol{e}$ contraintes *; chaque contrainte* $C \in \mathscr{C}$ *implique au plus* $\boldsymbol{k}$ *variables* $\mathrm{vars}(C) \subseteq \mathscr{X}$ *et définit la relation entre ces variables.*

Une *affectation* d'un ensemble de variables consiste à associer une *valeur* à chaque variable prise dans son domaine. En d'autres termes, le domaine d'une variable $\mathrm{dom}(X)$ définit l'ensemble des affectations *valides* de $X$. Dans cet article, nous considérons des domaines discrets, comportant au plus $\boldsymbol{d}$ valeurs.

La relation imposée par une contrainte $C$ définit l'ensemble d'au plus $\boldsymbol{\lambda}$ affectations *autorisées* notée $\mathrm{tab}(C)$ des variables $\mathrm{vars}(C)$. On définit usuellement la *dureté* d'une contrainte (*tightness*), comme la proportion du nombre d'affectations interdites par la contrainte par rapport au nombre d'affectations valides (i.e., le produit cartésien des domaines) de ses variables. Dans cet article, il nous sera plus utile de travailler avec son opposé, la *lâcheté* d'une contrainte (*looseness* ou *taux de remplissage*), proportion d'affectations autorisées par rapport aux affectations valides. On note $\boldsymbol{l}$ la lâcheté maximale d'une contrainte : on a donc $\lambda \leq ld^k$.

Le problème de satisfaction de contraintes (CSP) consiste à décider si une solution au réseau de contraintes (c'est-à-dire une affectation de toutes les variables satisfaisant toutes les contraintes) existe. Si toutes les contraintes peuvent être vérifiées en temps et en espace polynomiaux, le CSP est NP-complet.

DÉFINITION 3 (Consistance d'arc (généralisée), support). — *Soit* $C$ *une contrainte et* $X \in \mathrm{vars}(C)$. *Le* support *d'une valeur* $v \in \mathrm{dom}(X)$ *pour* $C$ *est une affectation valide de* $\mathrm{vars}(C)$ *autorisée par* $C$ *qui affecte* $v$ *à* $X$. $v$ *est* arc-consistante *(AC) pour* $C$ *ssi il existe un support de* $v$ *pour* $C$. $C$ *est AC ssi* $\forall X \in \mathrm{vars}(C)$, $\forall v \in \mathrm{dom}(X)$, $v$ *est AC pour* $C$.

Appliquer la consistance d'arc sur une contrainte $C$ consiste à supprimer toutes les valeurs du domaine des variables $\mathrm{vars}(C)$ qui ne sont pas AC pour $C$. On appelle cette procédure une *révision* de contrainte. Appliquer la consistance d'arc sur un réseau de contraintes consiste à réviser toutes les contraintes non AC jusqu'au point fixe.

Dans la suite de cet article, on considèrera à fin de simplification que toutes les contraintes sont définies sous la forme d'une liste de $k$-uplets *autorisés*. Il reste toujours possible d'utiliser plusieurs types de contraintes dans le même réseau, en associant à chacune un algorithme de filtrage différent.

Les structures de données sont définies à partir d'ensembles, tableaux (ou toute autre structure indexée à partir de 1) et de séquences (généralement implantées par des listes chainées). Pour un tableau $T$, $T[i]$ représente le $i^{\mathrm{e}}$ élément du tableau. Pour une séquence $S = \langle 1, 2, 3 \rangle$, on définit l'opérateur d'ajout en tête :: tel que $0 :: S = \langle 0, 1, 2, 3 \rangle$. Cet opérateur permet également de « déconstruire » les séquences selon

**Algorithme 1 :** AC($\mathcal{N}$, $Q$)

**Données** : Un réseau de contraintes $\mathcal{N}$ et l'ensemble $Q$ des contraintes potentiellement non AC. $\forall C \in Q$, $\mathit{modif}(C)$ contient la liste des variables modifiées depuis la dernière révision de $C$. $\forall C \notin Q$, $\mathit{modif}(C) = \emptyset$.

**Résultat** : Le réseau $\mathcal{N}$ dont les valeurs non AC ont été supprimées.

```
1  si Q = ∅ alors retourner N
2  sinon
3      Choisir C dans Q
4      Δ ← C.revise(modif(C))
5      si Δ = ⊥ alors retourner ⊥
6      sinon
7          modif(C) ← ∅
8          pour chaque Y ∈ Δ, C' ∈ 𝒞 − C | Y ∈ vars(C') faire
9              modif(C') ← modif(C') ∪ {Y}
10     Q' ← Q ∪ {C' ∈ 𝒞 | vars(C') ∩ Δ ≠ ∅} − C
11     retourner AC(N, Q')
```

la notation classique en programmation fonctionnelle : $h :: T \leftarrow S$ permet d'obtenir $h = 1$ et $T = \langle 2, 3 \rangle$.

## 3. Algorithmes de base

On considère que le CSP est résolu par l'algorithme MAC : recherche arborescente systématique en profondeur d'abord, avec maintien de la consistance d'arc à chaque nœud (Sabin & Freuder, 1994). L'algorithme 1 établit la consistance d'arc à l'aide d'une file de propagation à gros grain, qui contient l'ensemble des contraintes à réviser. La révision d'une contrainte $C$ est réalisée par un appel à la fonction `revise` rattachée à $C$ (ligne 4). Cette fonction renvoie la liste des variables modifiées, ou la valeur $\bot$ si une inconsistance, i.e., un domaine ou une table vide, a été détectée. La file de propagation est alors mise à jour en conséquence : il faudra à nouveau réviser les autres contraintes impliquant les variables modifiées (ligne 10). De plus, l'algorithme maintient, pour chaque contrainte, la liste $\mathit{modif}$ des variables modifiées depuis la dernière révision de celle-ci (lignes 7 à 9), comme proposé par Boussemart et al. (2004).

On révise une contrainte en recherchant un support pour chaque valeur : les valeurs n'ayant pas de support sont supprimées. L'algorithme, trivial, n'est pas présenté en extension dans cet article. L'algorithme 2 réalise la recherche des supports dans une table. On parcourt la liste des $k$-uplets autorisés (ligne 3) et vérifie leur validité (ligne 4). $\tau[X]$ est la valeur affectée à la variable $X$ dans le $k$-uplet $\tau$. Enfin, l'algorithme complète la liste des valeurs supportées (lignes 5 et 6). Lecoutre (2011) propose une optimisation à STR sous la forme de la structure $\mathit{Seek}$, que l'on peut appliquer dès

**Algorithme 2 :** seekSupports($C$)

**Données** : Une contrainte $C$
**Résultat** : Une application $Supp : X \rightarrow \{\text{valeurs}\}$ qui associe à chaque variable $X$ de $\text{vars}(C)$ l'ensemble des valeurs actuellement supportées par la contrainte

```
1  ∀X ∈ vars(C), Supp(X) ← ∅
2  Seek ← vars(C)
3  pour chaque τ ∈ tab(C) faire
4      si ∀X ∈ vars(C), τ[X] ∈ dom(X) alors
5          pour chaque X ∈ Seek faire
6              Supp(X) ← Supp(X) ∪ τ[X]
7              si Supp(X) = dom(X) alors
8                  Seek ← Seek − X
9                  si Seek = ∅ alors retourner Supp
10 retourner Supp
```

ce stade : il est inutile de traiter les variables pour lesquelles toutes les valeurs ont déjà été marquées comme supportées. Si toutes les valeurs de toutes les variables sont déjà supportées, on peut interrompre l'algorithme (lignes 7 à 9).

Lors de la recherche d'une solution au CSP, on effectue la recherche de supports pour une contrainte $C$, à chaque fois qu'une valeur est supprimée du domaine d'une variable concernée par $C$, c'est-à-dire jusqu'à $kd$ fois tout au long d'une branche de l'arbre de recherche. L'algorithme 2 a une complexité temporelle en $O(k\lambda)$ et ne présente aucune propriété d'incrémentalité : le filtrage a donc une complexité en $O(ek^2d\lambda)$ pour une branche de l'arbre de recherche.

## 4. L'algorithme *STR(2)* (*Simple Table Reduction*)

STR a été initialement proposé par Ullmann (2007). L'idée de STR est de ne conserver entre deux appels à l'algorithme 2 que la liste des $k$-uplets valides, c'est-à-dire vérifiant la condition de la ligne 4. Ainsi, un $k$-uplet détecté invalide une fois ne sera plus considéré dans la même branche de l'arbre de recherche. La contribution de Ullmann est une technique permettant de « filtrer » la table et de la restaurer en temps constant en cas de retour-arrière. Lecoutre (2011) améliore l'algorithme (sous le nom *STR2*), notamment en découplant dans l'algorithme 2 l'opération de filtrage de la table (ligne 4) de la validation des supports (lignes 5 à 9). Cette stratégie permet de ne contrôler la validité des $k$-uplets qu'au niveau des variables modifiées.

La figure 1 montre la technique utilisée par *STR* pour maintenir la version filtrée de la liste de $k$-uplets, et surtout la restaurer en temps constant. On utilise l'algorithme 3 : son paramètre $Mod$ fournit la liste des variables modifiées depuis le dernier appel de

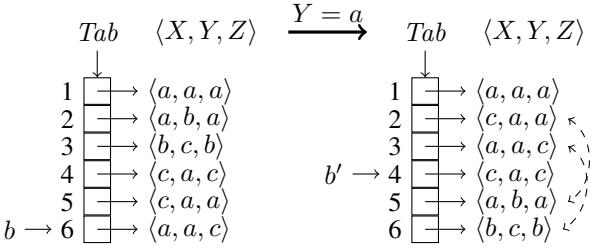



FIGURE *1. Filtrage d'une table* $(Tab, b)$ *par* STR

**Algorithme 3 :** filterSTR($Tab$, $b$, $Mod$)

**Données** : Une table $Tab$ sous la forme d'un tableau de $k$-uplets autorisés potentiellement non valides, une borne numérique $b$ et une séquence de variables modifiées $Mod$

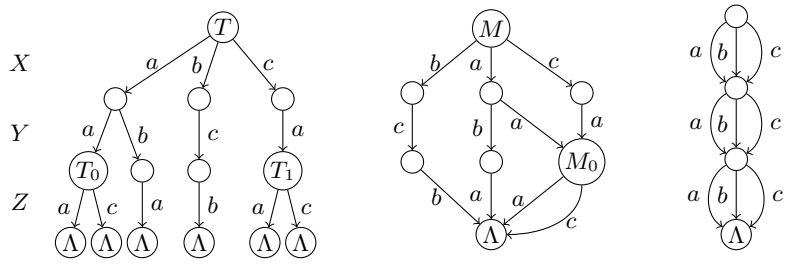
**Résultat** : La table $Tab$ réordonnée, et une nouvelle borne $b'$ telle que tous les $k$-uplets de $Tab[1]$ à $Tab[b']$ sont valides

```
1 b' ← b
2 pour i ← b à 1 faire
3     si ∃X ∈ Mod | Tab[i][X] ∉ dom(X) alors
4         échanger Tab[i] et Tab[b']
5         b' ← b' − 1
6 retourner (Tab, b')
```

la fonction. La table est représentée par un couple $(Tab, b)$ où $Tab$ est un tableau de $k$-uplets et $b$ est un nombre entier représentant l'index du dernier $k$-uplet valide dans le tableau (initialement, $b = \lambda$). Pour supprimer un $k$-uplet de la structure, on le permute avec $Tab[b]$, puis on décrémente $b$ (lignes 4 et 5 de l'algorithme 3). Il suffit de sauvegarder $b$ pour être capable de restaurer la structure à un état antérieur.

EXEMPLE 4. — Dans l'exemple de la figure 1, les $k$-uplets 2 et 3 ne sont plus valides après l'affectation $Y = a$ : ils sont respectivement permutés avec les $k$-uplets 6 et 5, et $b$ est décrémenté deux fois. Sa nouvelle valeur $b'$ indique que ces $k$-uplets ne doivent plus être pris en compte. Si $b$ est restauré à son ancienne valeur 6, les $k$-uplets supprimés sont immédiatement disponibles à nouveau. □

Comme chaque variable ne peut être modifiée que $d$ fois, la complexité amortie du filtrage de la table par STR2 tout au long d'une branche d'un arbre de recherche est en $O(kd\lambda)$, ce qui représente un facteur $k$ par rapport à *STR*. Ce filtrage étant réalisé, on peut appliquer l'algorithme 2 sans avoir à faire le test de la ligne 4. Malheureusement, le reste de la fonction ne montre pas de propriété d'incrémentalité théorique dans le pire cas : la complexité globale du filtrage reste donc inchangée. Cependant, nous nous sommes rapidement rendu compte que le temps passé à filtrer est en moyenne bien plus important que le temps passé à marquer les valeurs supportées, grâce no-




(a) Trie (b) MDD (c) MDD « complet »

FIGURE 2. *Un Trie $T$ et un MDD $M$ représentant la table de l'exemple 1, et un MDD représentant la table « complète » $\{a, b, c\}^3$. $T_0$ et $T_1$ sont identiques : ils sont fusionnés en un seul nœud $M_0$ dans le MDD correspondant.*

tamment aux optimisations apportées par la structure $Seek$. C'est bien le contrôle de validité qui fait office de goulot d'étranglement de l'algorithme en pratique. À titre d'illustration, la résolution d'un problème aléatoire de modèle RD (Xu, Boussemart, Hemery & Lecoutre, 2007) avec $(n, d, k, e, \lambda) = (20, 5, 5, 75, 2\,031)$ ($l = 65\,\%$) nécessite environ 140 s pour parcourir 213 099 nœuds de recherche dans nos conditions d'expérimentation (cf. section 8). L'outil d'échantillonnage inclus dans la JVM nous apprend qu'environ 90 s sont utilisées à effectuer les tests de validité (12 s à parcourir la structure et 78 s à contrôler les domaines), et seulement 14 s à marquer les supports. Les 34 s restantes concernent le chargement du problème, le calcul des heuristiques, la gestion de la file de propagation ou encore la suppression et la restauration des valeurs des domaines au cours de la recherche, et sont donc hors du champ de l'étude.

La dernière amélioration de l'algorithme, ***STR3*** (Lecoutre et al., 2012), atteint l'optimal $O(e(k\lambda + m))$ le long d'une branche de l'arbre de recherche de profondeur $m$. en indexant les $k$-uplets (chaque valeur des variables pointe sur les $k$-uplets où elle apparait) et en générant des listes de dépendances. ***STR3*** est considérablement plus compliqué à mettre en œuvre que les précédentes versions de l'algorithme et nécessite l'utilisation d'une file de propagation à grain fin. Mairy et al. (2014) ont récemment proposé une famille d'algorithmes, dont certains théoriquement optimaux, pour la propagation de contraintes *table* : ***AC5TC***. Cependant, ces algorithmes nécessitent une file de propagation à grain fin et ne permettent pas de traiter efficacement les contraintes sans structure, ou au contraire très structurées (e.g., sous forme de MDD). L'algorithme ne semble également réellement efficace que sur des contraintes d'arité limitée, inférieure à 5.

## 5. Tries et MDD

Le Trie (pour *Re*trie*val*) est une structure de données arborescente conçue pour rechercher une chaine de caractères dans un dictionnaire avec un comportement théo-

rique meilleur qu'une table de hachage (Fredkin, 1960). On peut l'utiliser pour rechercher un $k$-uplet dans une table en $O(k)$. On le définit alors de la manière suivante :

DÉFINITION 5 (Trie). — *Un Trie $T$ représentant une table $Tab$ peut être soit*

– *une feuille $\Lambda$ représentant l'ensemble constitué du $k$-uplet vide $\{\langle\rangle\}$, soit*

– *une application $T : i \to T'$ telle que $T'$ est un Trie qui représente l'ensemble des $(k-1)$-uplets $\{\tau, \ldots\}$ tels que $(i :: \tau) \in Tab$.*

$T$ forme un arbre de $\nu_{\text{trie}}$ arcs, comme sur la figure 2a. Un nœud donné représente tous les $k$-uplets commençant par les mêmes valeurs (i.e., le même préfixe). En ce sens, le Trie réalise une *compression* de la table. Chaque feuille du Trie représente un $k$-uplet de la table d'origine. Dans le pire des cas, aucun préfixe commun ne peut être identifié : toutes les premières valeurs de chaque $k$-uplet sont différentes. Dans ce cas, le Trie contient $k\lambda$ arcs. Cela nécessite de très grands domaines et/ou des taux de remplissage très faibles ($d \geq \lambda$). On peut également montrer que $\nu_{\text{trie}}$ est en $O(d^k)$, contre $O(k\lambda)$ ou $O(kd^k)$ pour une table. Quand le taux de remplissage est important ($d$ et $k \ll \lambda$) la compression réalisée est d'autant plus intéressante.

Un MDD est une généralisation du Trie pour lequel on autorise que des sous-arbres identiques soient regroupés. La représentation d'un MDD est donc un DAG à une racine et $\nu_{\text{mdd}}$ arcs (cf figure 2b). Pour une même table, on a donc $\nu_{\text{mdd}} \leq \nu_{\text{trie}} \leq k\lambda$. Pour construire le MDD à partir d'une table quelconque, on commence par construire le Trie correspondant, puis on parcourt celui-ci en indexant chaque nœud ; le calcul de l'index se fait généralement en $O(d)$. Quand on rencontre un nœud tel qu'un nœud équivalent a déjà été indexé, il peut être supprimé et remplacé par le nœud indexé : cf la fonction `mddReduce` de Cheng et Yap (2010). L'opération complète se fait en $O(k\lambda + d\nu_{\text{trie}})$. Le MDD d'une table structurée peut souvent être construit directement par une fonction ad-hoc, mais le principe de l'indexation reste présent. On atteint alors généralement une complexité en $O(\nu_{\text{trie}})$.

Ainsi, la table de l'exemple 1 comprend 6 triplets, pouvant être représentés par une matrice de 18 éléments, le Trie de 12 arcs de la figure 2a ou encore le MDD de 11 arcs de la figure 2b. Le pire des cas pour la matrice est illustré par une table « complète » des 27 triplets sur les trois valeurs $a$, $b$ et $c$, qui peut être représenté par une matrice de 81 éléments, un Trie de $3^1 + 3^2 + 3^3 = 39$ arcs. Il s'agit d'un cas favorable pour le MDD, qui ne nécessite que 9 arcs (figure 2c). Différents ordonnancements des variables peuvent conduire à des Tries et MDD de tailles différentes. Trouver un ordonnancement de variables conduisant au Trie ou MDD de taille minimale est un problème NP-difficile (Bollig & Wegener, 1996) qui ne sera pas traité dans cet article.

L'utilisation de Tries et MDD pour représenter des contraintes *table* a été envisagée, notamment dans les travaux de Gent et al. (2007) pour les Tries et de Cheng et Yap (2010) pour les MDD. Les MDD sont une solution plus robuste. Le nombre de nœuds est moindre que dans un Trie ce qui constitue une économie intéressante en espace mémoire et en temps de traitement. Cheng & Yap proposent *MDDC*, un algorithme qui parcourt un MDD à la recherche de valeurs supportées, en sauvegardant les parties démontrées comme valides ou invalides. Ceci permet d'éviter du travail redon-

dant. Deux mécanismes sont utilisés pour sauvegarder les résultats d'une recherche : un *timestamp* associé à chaque nœud assure de ne pas parcourir celui-ci deux fois au cours de la même exécution et un *sparse set* (Briggs & Torczon, 1993) enregistre les nœuds invalides d'une exécution à l'autre de l'algorithme. Les *sparse sets* peuvent être « sauvegardés » et « restaurés » en temps constant en cas de retour-arrière. Il est intéressant de noter que la technique utilisée par les *sparse sets* coïncide avec la technique de STR.

**Algorithme 4 :** MDDC($M$, $ts$, $invalid$, $p$, $Supp$)

**Données** : Un MDD $M$ portant sur les variables $\mathcal{X}$, un *timestamp* $ts$ tel que $Ts[M] = ts$ ssi $M$ a déjà été parcouru et autorise au moins un support valide, l'ensemble $invalid$ des MDD tels que tous les supports qu'ils autorisent ne sont pas valides, le niveau $p$ du MDD ($\mathcal{X}[p]$ est la variable correspondant au niveau courant), et $Supp$ l'application associant à chaque variable $X \in \mathcal{X}$ la liste des valeurs supportées. $\delta$ indique le niveau à partir duquel toutes les valeurs de toutes les variables sont supportées (initialement $|\mathcal{X}| + 1$)

**Résultat** : **true** ssi le MDD autorise au moins un support valide, **false** sinon.

```
1  si M = Λ ∨ ts = Ts(M) alors retourner true
2  sinon si M ∈ invalid alors retourner false
3  sinon
4      X ← 𝒳[p]
5      valid ← false
6      pour chaque i ∈ dom(X) | MDDC(M[i], ts, invalid, p + 1, Supp) faire
7          valid ← true
8          Supp(X) ← Supp(X) ∪ {i}
9          si p + 1 = δ ∧ Supp(X) = dom(X) alors
10             δ = p
11             break;
12     si valid alors Ts(M) ← ts
13     sinon invalid ← invalid ∪ {M}
14     retourner valid
```

L'algorithme 4 ci-dessus est une version légèrement simplifiée de l'algorithme de Cheng et Yap (2010). La variable $\delta$ et les lignes 8 à 10 implantent l'*early cutoff optimisation*, au fonctionnement similaire à la structure $Seek$ de notre algorithme 2 : l'idée est d'interrompre l'algorithme si toutes les valeurs des variables situées plus bas dans le MDD ont déjà été marquées comme supportées, et que le MDD autorise au moins un support valide.

EXEMPLE 6. — Simulons l'exécution de MDDC sur le MDD de la figure 2b. La ligne 5 de l'algorithme 4 construit l'ensemble des valeurs supportées de la variable $X$ par des appels récursifs à MDDC. L'algorithme parcourt d'abord la branche de gauche. Lorsqu'on atteint la feuille, on sait que la branche parcourue constitue un $k$-uplet

autorisé et valide, ce qui autorise à marquer toutes les valeurs rencontrées comme supportées ($X = b$, $Y = c$ et $Z = b$). L'algorithme parcourt ensuite la branche centrale qui permet de marquer les valeurs $X = a$, $Y = b$, $Z = a$, puis reprend à $Y = a$, atteignant alors le sous-graphe $M_0$. Celui-ci permet de marquer $Y = a$, $Z = a$ (qui avait déjà été marquée) et $Z = c$. Enfin, la branche de droite permet de marquer $X = c$ et $Y = a$ (à nouveau). $M_0$ a déjà été parcouru depuis une autre branche et a été reconnu valide. Il n'est donc pas exploré une deuxième fois.

Si, suite à l'affectation $Y = a$, la structure est à nouveau explorée, la première branche ($X = b$) devient une impasse ne permettant pas d'atteindre une feuille : elle est marquée comme invalide dans le *sparse set* et ne sera plus explorée tant que le domaine de $Y$ et le *sparse set* n'auront pas été restaurés à un état antérieur. □

Le *early cutoff optimization* n'apporte pas un grain aussi fin que la structure $Seek$ utilisée par STR2. Cependant, comme dans le cas de *STR*, il est possible dans MDDC de découpler la validation du MDD du marquage des valeurs supportées afin d'exploiter les informations concernant les variables modifiées : il inutile de contrôler la présence des valeurs dans le domaine des variables non modifiées. Nous avons implémenté et testé cette version, mais avec des résultats pratiques finalement mitigés. En effet, le découplage des opérations de filtrage et de marquage des valeurs supportées nécessite de contrôler deux fois la validité de certains arcs : par exemple, toujours sur la figure 3, l'arc $X = b$ sera parcouru pour constater l'invalidité sauvegardée de $M_3$, et la validité de $Y = b$ sera également contrôlée pour constater que cette partie du MDD n'est plus utilisée.

Ces limitations rendent malheureusement *MDDC* particulièrement peu efficace par rapport à *STR2* sur des tables peu structurées (par exemple générées aléatoirement). Dans la suite de cet article, nous nous intéresserons plus particulièrement aux MDD. Nous proposons un nouvel algorithme, inspiré de *STR2* mais basé sur des MDD. Cet algorithme développe plus efficacement les possibilités d'incrémentalité exploitables sur une telle structure.

## 6. MDDF (MDD-Filtering)

Notre proposition s'inspire des structures de données persistantes (Driscoll, Sarnak, Sleator & Tarjan, 1989), parfois dites immuables et notamment utilisées dans les langages de programmation fonctionnels comme ML, Haskell ou Scala (Odersky et al., 2001–2014). Ces langages découragent ou interdisent la modification des données au cours de l'exécution des programmes, ce qui encourage ou impose ce type de structure. Le principe général est que pour ajouter un élément à une structure de données, on crée une copie de la précédente à laquelle on ajoute l'élément. Quand la structure prend la forme d'un graphe acyclique, il est généralement possible de réutiliser une grande partie du graphe original en pointant vers des parties du graphe non impactées par la modification. Dans l'exemple le plus simple, une liste simplement chainée est définie par un élément de tête et une liste de queue. Ajouter un élément se fait en générant une nouvelle liste telle que la queue est une référence à l'ancienne liste.

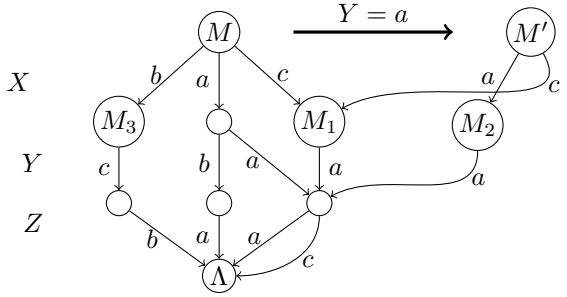



FIGURE *3. Filtrage d'un MDD immuable.*

Comme on interdit de modifier le contenu des listes, il ne pourra pas y avoir d'effets secondaires (*side effects*) les modifiant de manière inattendue.

La plupart des structures de données et algorithmes usuels ont un équivalent persistant dont les opérations ont la même complexité temporelle que sur la structure modifiable correspondante. Les structures persistantes simplifient considérablement la conception et le débogage des algorithmes (en bannissant les effets secondaires), mais elles permettent surtout de s'affranchir de toute gestion de la synchronisation lors de la parallélisation. Dans le cas des algorithmes d'exploration systématiques comme MAC, qui nous intéresse dans cet article, ces structures de données sont précieuses dans la mesure où elles permettent de garder une trace de l'évolution des données et ainsi effectuer facilement des retour-arrière.

Une version légèrement dégradée de *STR(2)* peut être implantée en utilisant une structure persistante simple : une liste chainée de $k$-uplets. Lorsque l'on filtre la table, on génère une autre liste ne comprenant que les $k$-uplets valides, et on conserve une référence sur la liste d'origine pour réaliser le retour-arrière en temps constant. Il n'y a pas de surcout de complexité temporelle, mais on doit allouer une cellule de liste chainée à chaque fois que l'on vérifie la validité d'un $k$-uplet, et la désallouer en cas de retour-arrière. Cette dernière opération est transparente pour le développeur si un ramasse-miettes est utilisé. En terme d'espace, on ajoute une liste chainée de taille en $O(\lambda)$ à chaque niveau de l'arbre de recherche. Il n'est pas nécessaire de dupliquer les $k$-uplets eux-mêmes, puisqu'ils sont immuables. Coder la phase de filtrage de *STR(2)* peut ainsi se faire de manière très élégante dans un langage fonctionnel.

L'idée centrale de notre contribution est d'appliquer un principe similaire à cette version « persistante » de *STR* aux MDD : la phase de filtrage au cœur du principe de *STR* est appliquée à un MDD. La fonction `filterMDD` renvoie une « copie » filtrée du MDD, tout en conservant telles quelles les parties non modifiées du MDD. La figure 3 illustre ce principe : obtenir une version du MDD d'origine tel que $Y = a$ ne nécessite que de créer les deux nœuds $M'$ et $M_2$. Par rapport à *STR(2)*, on bénéficie de toute la puissance de la compression apportée par les MDD. Ainsi, réduire le MDD après une modification de la première variable $X$ ne nécessite que de parcourir

**Algorithme 5 :** filterMDD($M, ts, Mod, \mathcal{X}$)

**Données** : Un MDD $M$ représentant un ensemble de $k$-uplets potentiellement non valides, un *timestamp* $ts$, une liste de variables modifiées $\mathrm{mod}$ , la liste des variables concernées par les $k$-uplets de $M$

**Résultat** : Un MDD égal à $M$ privé de tous ses $k$-uplets non-valides

```
1 si Mod = ∅ alors retourner M
2 sinon si ts ≠ Ts(M) alors
3 |   Ts(M) ← ts
4 |   M' ← nouveau tableau de d MDD vides
5 |   X :: X' ← X
6 |   pour chaque M[i] non vide | i ∈ dom(X) faire
7 |   |   M'[i] ← filterMDD(M[i], ts, Mod − X, X')
8 |   Last(M) ← M'
9 retourner Last(M)
```

et générer un seul nouveau nœud. Le gain potentiel par rapport à *STR* est de $O(\lambda)$ opérations sans aucun surcout théorique en termes de complexité temporelle (avec cependant une restriction par rapport à *STR2*, discutée ci-après).

Le cœur de notre contribution est l'algorithme 5. La fonction `filterMDD` prend en argument un MDD $M$, un *timestamp* $ts$, l'ensemble $Mod$ des variables modifiées depuis le dernier appel à l'algorithme (maintenu par l'algorithme 1), et la séquence $\mathcal{X}$ des variables impliquées par la contrainte, dans l'ordre qui a servi à la génération du MDD.

La condition d'arrêt de la fonction récursive est simple : le MDD est inchangé si aucune variable n'a été modifiée (ligne 1). À chaque fois qu'une variable est traitée, elle est retirée de l'ensemble (appel de la ligne 7), ce qui assure que l'algorithme saura s'interrompre une fois toutes les variables modifiées traitées. Le *timestamp* sert à ne parcourir chaque nœud qu'une seule fois. Il est comparé à une valeur $Ts(M)$ associée à chaque nœud. Quand le nœud est traité, on réaffecte $Ts(M)$ à la nouvelle valeur de $ts$ : si le même *timestamp* est à nouveau rencontré, il s'agit du même nœud que précédemment et on peut renvoyer le même résultat $Last(M)$ qu'à la précédente exécution (lignes 2, 3, 8).

Le premier élément $X$ de $\mathcal{X}$ (ligne 5), correspond à la variable à traiter au niveau courant. Notons que si $X \notin Mod$, il est inutile de contrôler la présence de chaque $i$ dans $\mathrm{dom}(X)$ à la ligne 6.

À chaque appel récursif de la fonction, on reconstitue un nouveau nœud $M'$ (lignes 5 et boucle des lignes 6–7) créé pour l'occasion. Les arcs pointant sur des MDD vides sont automatiquement supprimés. Si le nouveau nœud $M'$ est identique à l'ancien $M$, il peut être libéré et $M$ est utilisé à sa place. Il est possible qu'après l'exécution de l'algorithme 5, d'autres nœuds se révèlent identiques. Par exemple sur la figure 3, les

nœuds $M_1$ et $M_2$ sont identiques après filtrage et pourraient être fusionnés. Cependant, le cas semble suffisamment rare pour que la réduction des MDD après chaque filtrage soit intéressante, ce qui a été confirmé par des tests préliminaires. Il s'agit probablement d'une voie à explorer.

EXEMPLE 7. — Voici l'exécution de l'algorithme 5 sur le MDD de la figure 3 après l'affectation de la valeur $a$ à la variable $Y$. La fonction est appelée avec $Mod = \{Y\}$ et $\mathcal{X} = \langle X, Y, Z\rangle$. $\forall M$, $Ts(M) = 0$ et $Last(M) = M$. $ts = 1$. On génère un nouveau nœud $M'$ initialement vide (ligne 4). L'appel récursif de la ligne 7 va contrôler le deuxième niveau du MDD pour remplir le nouveau nœud, avec $\mathcal{X} = \langle Y, Z\rangle$.

Pour $M[a]$, l'appel récursif va générer le nœud $M_2$. L'arc $M_2[a]$ est valide, on appelle alors récursivement `filterMDD` avec $Mod = \emptyset$. La condition d'arrêt est rencontrée, et le nœud est renvoyé tel quel (en effet, il est inutile de contrôler la validité du domaine de la variable $Z$). L'arc $M_2[b]$ n'est pas valide : $M_2$ n'aura donc qu'un seul fils. Après la boucle, on a donc $Last(M[a]) = M_2$, qui est renvoyé.

Pour $M[b]$ ($M_3$), aucun arc n'est valide. Aucun appel récursif n'est effectué. Le nœud reste vide. La valeur renvoyée est $Last(M_3) = \emptyset$.

Enfin, pour $M[c]$ ($M_1$), on contrôle la validité de $M_1[a]$, qui conduit à un appel récursif avec $Mod = \emptyset$, comme pour $M'[a]$. Le nœud obtenu est identique au nœud initial : il est ignoré et on conserve $Last(M_1) = M_1$, qui est renvoyé.

À l'issue de l'exécution de l'algorithme, on obtient le nouveau nœud $M'$ à partir de $M$, tel que $M'[a] = M_2$, $M'[b] = \emptyset$ et $M'[c] = M_1$, inchangé par l'affectation. □

L'algorithme 6 recherche ensuite un support pour chaque valeur en parcourant le MDD : cette partie de l'algorithme est très proche de *MDDC*, à ceci près que grâce à la phase de filtrage préalable, on sait que toutes les valeurs apparaissant dans le MDD sont supportées, ce qui permet quelques optimisations. Les paramètres sont le MDD $M$, un *timestamp* $ts$ (différent du *timestamp* de l'algorithme 5), l'application $Supp$ associant à chaque variable l'ensemble des valeurs supportées (initialement vide), la liste des variables impliquées par la contrainte dans l'ordre ayant servi à générer le MDD $\mathcal{X}$ et enfin la liste des variables pour lesquelles il y a encore des valeurs sans support trouvé $Seek$. Ici encore, on utilise les *timestamps* pour éviter de traiter plusieurs fois le même nœud (lignes 1 et 2). L'ensemble $Seek$ permet d'arrêter le parcours si un support pour chaque valeur des variables situées plus bas dans le MDD a déjà été trouvé (lignes 5, 7 et 8). Ceci implémente un équivalent au *early cutoff optimization* de *MDDC*. À la fin de l'exécution de l'algorithme, l'ensemble $Seek$ ne contient que les variables pour lesquelles des valeurs n'ont pas de support : seules ces variables doivent être filtrées.

Pour optimiser l'exécution de la boucle de la ligne 6 de l'algorithme 5 ou de la ligne 4 de l'algorithme 6, notre implantation maintient la liste des indices pour lesquels $M[i]$ n'est pas vide en utilisant une technique proche de STR. Cette optimisation empêche de partager des nœuds entre plusieurs MDD. L'implantation n'est plus non plus strictement persistante. Une amélioration de nos implémentations sur ce point

**Algorithme 6 :** seekSuppMDD($M$, $ts$, $Supp$, $\mathcal{X}$, $Seek$)

**Données** : Un MDD $M$ représentant un ensemble de $k$-uplets valides, un *timestamp* $ts$, une application $Supp$ associant à des variables l'ensemble des valeurs connues comme supportées, la liste $\mathcal{X}$ de toutes les variables concernées par les $k$-uplets de $M$, et une liste $Seek \subseteq \mathcal{X}$ de variables contenant encore des valeurs potentiellement non supportées.

**Résultat** : L'application $Supp$ associant à chaque variable de $\mathcal{X}$ la liste de *toutes* les valeurs supportées, et la liste $Seek$ de *toutes* les variables contenant des valeurs non supportées.

```
1  si ts ≠ Ts(M) alors
2      Ts(M) ← ts
3      X :: X' ← X
4      pour chaque M[i] non vide faire
5          si X ∩ Seek = ∅ alors break
6          Supp(X) ← Supp(X) ∪ {i}
7          si Supp(X) = dom(X) alors
8              Seek ← Seek − X
9          (Supp, Seek) ← seekSuppMDD(M[i], ts, Supp, X', Seek)
10 retourner (Supp, Seek)
```

est à l'étude. D'autre part, une conception orientée objets nous permet de définir une implantation spécifique et optimisée des MDD vide et feuille (objets singleton), ainsi que des MDD à un ou deux fils.

## 7. Complexités et discussion

PROPRIÉTÉ 8. — *La complexité temporelle dans le pire des cas de l'algorithme 5 (filtrage du MDD) est en* $O(\nu_{\mathrm{mdd}})$.

Au cours de la procédure de filtrage, chaque arc du MDD est considéré au maximum une fois : la complexité de l'opération de filtrage est donc en $O(\nu_{\mathrm{mdd}})$. On remarque qu'il n'est pas possible d'ignorer complètement les niveaux intermédiaires auxquels les variables n'ont pas été modifiées. Si des variables ont été modifiées plus bas dans l'arbre, il faut quand même traverser ces nœuds (boucle de la ligne 12). Les propriétés d'incrémentalité de *MDDF* ne sont donc pas aussi fines que celles de *STR2*. Cependant, les *vérifications de domaine* ne sont faites qu'aux niveaux où les variables ont été modifiées. Comme chaque variable ne peut être modifiée que $d$ fois, chaque arc ne va être contrôlé qu'au maximum $d$ fois tout au long d'une branche de l'arbre de recherche.

PROPRIÉTÉ 9. — *La complexité temporelle dans le pire des cas de l'algorithme 6 (marquage des supports) est en* $O(\nu_{\mathrm{mdd}})$.

Le test à la ligne 5 de l'algorithme 6 peut être réalisé en temps amorti $O(k+\nu_{\mathrm{mdd}})$ au cours d'une exécution de l'algorithme en maintenant la position de la dernière variable présente dans $Seek$ (dans l'ordre défini par la contrainte). Les autres opérations considèrent chaque arc au plus une fois, on a $k \leq \nu_{\mathrm{mdd}}$ si $\lambda > 0$ (lorsque $\lambda = 0$, le réseau de contraintes est trivialement inconsistant), d'où la propriété.

PROPRIÉTÉ 10. — *La complexité temporelle dans le pire des cas du filtrage par consistance d'arc (algorithme 1) tout au long d'une branche de l'arbre de recherche si toutes les contraintes sont filtrées par* MDDF *est en* $O(ekd\nu_{\mathrm{mdd}})$.

Aucune propriété théorique d'incrémentalité n'a été observée sur les algorithmes 5 ni 6. Dans le pire des cas, chaque appel à l'algorithme concerne la modification de la variable située le plus bas dans le MDD. Ceci nécessite de parcourir tous les arcs du graphe. Cependant, le nombre de *vérifications de domaine* effectuées par *MDDF* est en $O(ed\nu_{\mathrm{mdd}})$ tout au long d'une branche de l'arbre de recherche. Après avoir filtré la table et recherché les supports, il faut également supprimer les valeurs sans support. Il est évident que ce cout est $O(nd)$ amorti sur toute la branche de l'arbre de recherche, puisque chaque valeur ne peut être supprimée qu'une fois. Si chaque variable est impliquée par au moins une contrainte, on a $n \leq ek$ : cet aspect est négligeable.

À titre de comparaison, la complexité temporelle dans le pire des cas de *STR2* tout au long d'une branche de l'arbre de recherche est en $O(ek^2d\lambda)$, celle de *MDDC* est $O(ekd\nu_{\mathrm{mdd}})$. *STR2* ne nécessite que $O(ekd\lambda)$ vérifications de domaine lors la phase de filtrage.

PROPRIÉTÉ 11. — *Si toutes les contraintes sont filtrées par* MDDF*, la complexité spatiale dans le pire des cas du filtrage par consistance d'arc tout au long d'une branche de l'arbre de recherche est* $O(ekd\nu_{\mathrm{mdd}})$.

Un MDD peut être filtré $kd$ fois (quand une variable est modifiée). Ceci peut, dans le pire des cas, entrainer à chaque fois la copie des $\nu_{\mathrm{mdd}}$ arcs du MDD.

L'algorithme *MDDC* de Cheng & Yap met en place une incrémentalité en enregistrant les nœuds valides et invalides. Un nœud est valide si et seulement si au moins un arc partant de ce nœud conduit à une feuille ou à un nœud valide. L'information concernant les nœuds invalides peut être conservée entre deux appels successifs. *MDDF* va plus loin. En enregistrant une copie du MDD, *MDDF* est capable d'éliminer chaque *arc* non valide. L'incrémentalité obtenue est donc bien meilleure. Les opérations de copie n'entrainent pas de surcout en termes de complexité temporelle.

Observons la figure 3 : une fois réalisé le filtrage suite à l'affectation $Y = a$, *MDDF* travaille sur le MDD $M'$, alors que *MDDC* reste sur le MDD original $M$. Le sous-MDD $M_3$ est marqué invalide, ce qui permet d'éviter de parcourir les deux arcs qu'il contient. Mais les arcs $X = b$ et $Y = b$ dans la branche centrale, sont toujours pris en compte par *MDDC*. Le surcout potentiel pour *MDDC* est de $d$ opérations sup-

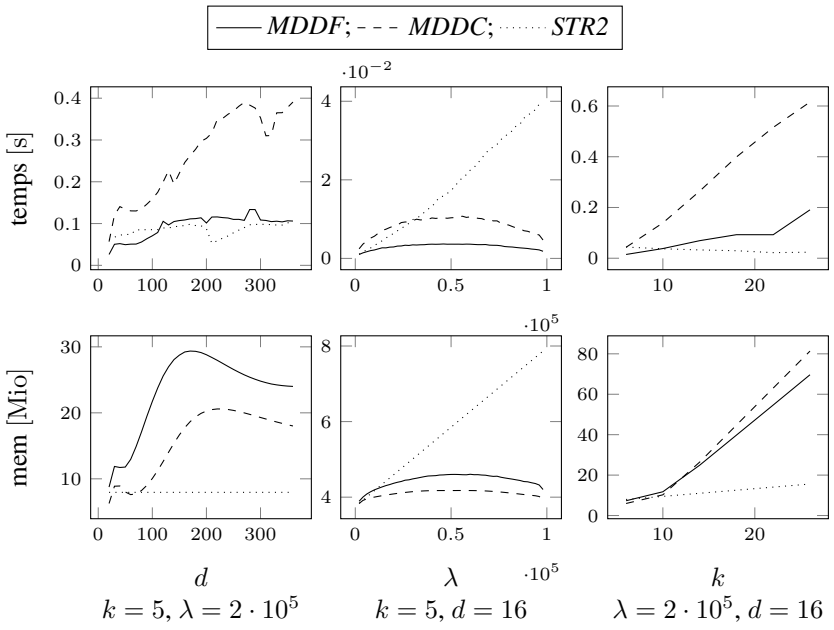



FIGURE *4. Temps et mémoire utilisée pour filtrer 20 niveaux d'une branche d'arbre de recherche simulée (moyennes sur 1 000 exécutions).*

plémentaires par nœud valide. Ceci a été constaté expérimentalement : cf la courbe en haut à gauche de la figure 4.

En plus de sa meilleure incrémentalité, *MDDF* permet d'exploiter plus finement les informations sur les variables modifiées fournies par l'algorithme de propagation. *MDDC* tel que décrit par Cheng & Yap n'exploite aucunement les informations concernant les variables modifiées, et nos expérimentations pour améliorer ce point (décrites à la fin de la section 5) ont été infructueuses. Dans le cas où seule la variable au sommet du graphe a été modifiée, *MDDF* permet de ne vérifier que $O(kd)$ valeurs alors que *MDDC* va potentiellement parcourir tout le graphe.

Finalement, *MDDF* nécessite d'instancier un tableau de taille $d$ à chaque parcours de nœud. Cela n'a pas d'impact sur la complexité temporelle, mais les allocations de mémoire peuvent avoir un cout non négligeable en pratique. L'impact sur la complexité spatiale n'est pas non plus neutre. Comme nous le verrons expérimentalement, si *MDDF* est presque toujours plus rapide que *MDDC*, il est surtout intéressant par rapport à *STR2* quand $\nu_{\mathrm{mdd}} \ll k\lambda$, ce qui implique des contraintes structurées et/ou un taux de remplissage ($l$) important.

## 8. Expérimentations

Les algorithmes ont été implantés en langage Scala 2.10 (Odersky et al., 2001–2014) sur notre solveur Concrete (Vion, 2006–2014). Ils sont exécutés sur une machine virtuelle Java 7.0u9 (OpenJDK 64 bit Server) avec 2 GiB de mémoire de tas allouée. Le système d'exploitation est basé sur un noyau Linux 3.8.3-x86_64 fonctionnant sur un processeur Intel Core i5-2500S CPU @ 2,7 GHz et disposant de 8 GiB de RAM. Les algorithmes de révision testés pour les contraintes d'arité supérieure à 2 définies en extension sont *STR2*, *MDDC* et *MDDF*. La stratégie de recherche est un arbre binaire basé sur une affectation des variables par $^{dom}/_{wdeg}$ et un ordre lexicographique décroissant pour les valeurs, avec résolution pseudo-aléatoire des égalités et redémarrages périodiques. La file de propagation des contraintes est ordonnée en FIFO. Notre implantation garantit que les arbres de recherche sont identiques quel que soit l'algorithme de révision utilisé.

Dans un premier temps, nous avons expérimenté les algorithmes sur des MDD générés aléatoirement, suivant une technique proposée par Cheng et Yap (2010), d'après six paramètres $(n, d, k, e, \lambda, q)$ : respectivement le nombre de variables $n$, de valeurs $d$, l'arité des contraintes $k$, leur nombre $e$, un nombre de tuples autorisés $\lambda$, et enfin un « taux de structure » $q$, qui définit la probabilité qu'a un nœud du MDD d'être dupliqué. Pour construire ces MDD, on parcourt un Trie « complet » de profondeur $k$ sur un alphabet de $d$ valeurs. Chaque feuille est conservée avec une probabilité $\lambda d^{-k}$, ou supprimée ($\lambda$ est donc une valeur *moyenne* ; la variance sera approximativement de $\lambda - \lambda^2 d^{-k}$). Enfin, chaque nœud est remplacé avec une probabilité $q$ par un autre nœud déjà généré.

La figure 4 montre l'évolution du temps de calcul et l'utilisation de la mémoire pour stocker une contrainte quand l'un des paramètres $d$, $\lambda$ ou $k$ augmente (on a gardé $q = 0$, ce qui signifie qu'on n'a pas dupliqué de nœud). Pour montrer l'impact de la création de nouveaux nœuds de MDD au cours de la recherche avec *MDDF*, une branche d'arbre de recherche est simulée en retirant au hasard 5 % des valeurs et en effectuant une révision à chaque nœud. La profondeur de la branche est limitée à 20. On remarque qu'avec les MDD, la mémoire atteint rapidement un maximum quand $\lambda$ augmente, et décroit quand $\lambda$ s'approche de $d^k$, là où une représentation en table voit sa taille augmenter linéairement. C'est un effet de la complexité en $O(d^k)$ du nombre d'arcs du MDD, et du cas très favorable des tables complètes pour le MDD. Cela laisse supposer que les algorithmes présentés restent performants même avec des taux de remplissage très importants. On constate que *MDDF* utilise plus de mémoire que *MDDC* au cours de la recherche, mais cela reste raisonnable : la mémoire utilisée par *MDDF* reste généralement inférieure à celle utilisée par *STR2* avec ces paramètres.

Nous avons souhaité observer l'évolution des performances en fonction du taux de remplissage $l$ sur un problème de base tel que $(d, k) = (5, 5)$, puis en augmentant séparément $d$ et $k$ à 8. Rappelons que $\lambda = ld^k$, $\nu_{\mathrm{mdd}} \leq k\lambda$ et que $\nu_{\mathrm{mdd}}$ est en $O(d^k)$. Le nombre de variables $n$ a été ajusté en conséquence pour que les problèmes restent de difficulté moyenne, c'est-à-dire nécessitant un temps de résolution médian

inférieur à 1 200 s pour tous les algorithmes avec $l = 80\,\%$ et $q = 0$, obtenu avec $n =$ respectivement 20, 13 et 13. Enfin, le rapport entre le nombre de contraintes et le taux de remplissage est ajusté de sorte que les problèmes soient proches de la transition de phase d'après les théorèmes du modèle RD (Xu et al., 2007). Pour une meilleure précision, nous avons fixé le nombre de contraintes et en avons déduit le taux de remplissage : $l = \exp(-e^{-1} n \ln d)$. Rappelons que pour que ces théorèmes s'appliquent, il faut que $l \geq k^{-1}$, ce qui fixe la limite basse de nos tests. Enfin, nous avons testé des problèmes sans aucune structure ($q = 0$) ou au contraire assez structurés ($q = 50\,\%$). Des valeurs indicatives de $e$, $\lambda$ et $\nu_{\mathrm{mdd}}$ sont précisées sur les graphes.

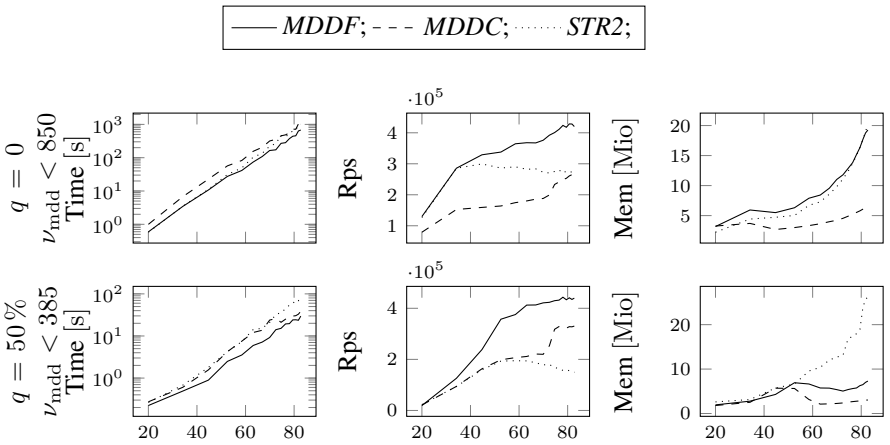
Les résultats sont présentés sur la figure 5. Répartir les $k$-uplets interdits sur un plus grand nombre de contraintes réduit la capacité de filtrage de celles-ci, ce qui augmente la taille des arbres de recherche de manière exponentielle. Cela rend les courbes des temps de résolution (à gauche) peu lisibles, nous conseillons d'observer en priorité le nombre de révisions par seconde, au centre. Rappelons que l'arbre de recherche ainsi que la séquence de révisions sont strictement identiques pour les trois versions présentées.

On constate que sur ces problèmes aléatoires, *MDDF* est toujours plus rapide que *MDDC*. Il est aussi plus rapide que *STR2* à l'exception des problèmes sans structure ayant une arité forte et un taux de remplissage faible ($k = 8$, $l < 20\,\%$, $q = 0$). C'est dans ces conditions que la limitation des possibilités d'incrémentalité de *MDDF* par rapport à *STR2* a le plus d'impact, que l'effet de la compression des MDD ne suffit plus à compenser. Sur des problèmes structurés, *STR2* est comme prévu le moins efficace. *MDDC* reste toujours le moins gourmand en espace mémoire, grâce à ses structures de données minimalistes.

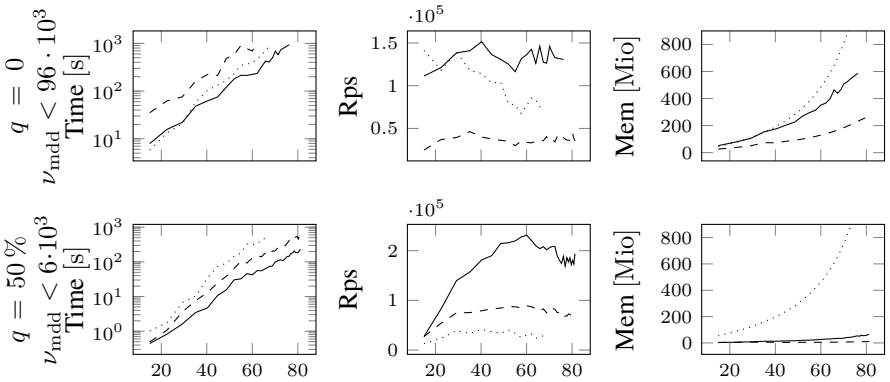
*Tableau 1. Résultats sur problèmes structurés : pour chaque classe d'instances, métriques principales (moyenne + 2,3 × écart-type), la vitesse de recherche en milliers de révisions par seconde, et la mémoire requise (en Mio, médianes).*

| | | | | | *MDDF* | | *MDDC* | | *STR2* | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $d$ | $k$ | $\lambda$ | $\nu_{\mathrm{mdd}}$ | trps | mem | trps | mem | trps | mem |
| *carseq-100* | 23 | 57 | $10^{22}$ | 2 132 | **184** | 8 | 101 | **3** | – | out |
| *carseq-200* | 24 | 80 | $10^{39}$ | 6 151 | **190** | 42 | 64 | **5** | – | out |
| *carseq-300* | 24 | 97 | $10^{52}$ | 7 302 | **162** | 71 | 65 | **8** | – | out |
| *carseq-400* | 24 | 111 | $10^{91}$ | 16 772 | **122** | 203 | 35 | **12** | – | out |
| *knapsack* | 13 | 201 | $10^{48}$ | 474 697 | **5** | 98 | 1 | **12** | – | out |
| *renault* | 20 | 9 | 17 733 | 876 | **29** | **7** | 25 | **7** | 26 | 15 |
| *crossword* | 26 | 15 | 36 192 | 39 557 | 35 | 20 | 17 | 8 | **52** | **5** |
| *tsp* | $10^3$ | 3 | 26 872 | 27 389 | 904 | 16 | 437 | **6** | **1 180** | 7 |
| *nonogram* | 2 | 36 | 426 | 249 | **244** | 24 | 114 | 24 | 200 | 24 |

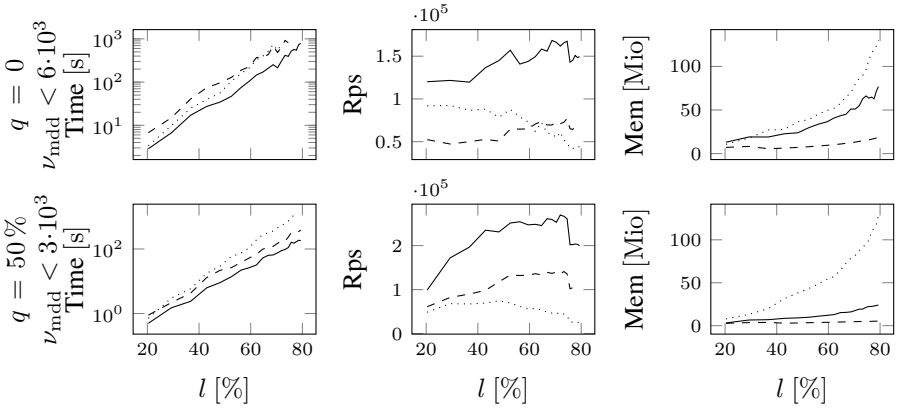
La table 1 présente des tests sur des problèmes plus structurés qui confirment les bons résultats de *MDDF*. Les problèmes *renault*, *crossword* et *tsp* sont issus de la troisième compétition internationale de solveurs CSP (van Dongen, Lecoutre & Roussel,




(a) $(n, d, k) = (20, 5, 5)$; $e = 20, \ldots, 170$; $\lambda = 625, \ldots, 3 \cdot 10^3$




(b) $(n, d, k) = (13, 5, 8)$; $e = 11, \ldots, 101$; $\lambda \approx 58 \cdot 10^3, \ldots, 318 \cdot 10^3$




(c) $(n, d, k) = (13, 8, 5)$; $e = 17, \ldots, 117$; $\lambda \approx 7 \cdot 10^3, \ldots, 26 \cdot 10^3$

FIGURE *5. Temps de résolution, vitesse de propagation (revisions par seconde) et mémoire utilisée pour résoudre des problèmes aléatoires. Chaque point est une médiane obtenue à partir de 25 problème générés et résolus.*

2008). Ce sont toutes les séries non aléatoires faisant intervenir des contraintes *table*. À titre d'indication, nous présentons les métriques principales caractérisant chaque problème $d$, $k$, $\lambda$ et $\nu_{\mathrm{mdd}}$. Le chiffre présenté pour chaque métrique est la moyenne additionnée de $2,3\times$l'écart-type : sous l'hypothèse que la répartition soit proche d'une loi normale, cela signifie que $99\,\%$ des problèmes ont des caractéristiques inférieures à la valeur indiquée. Par exemple, $99\,\%$ des problèmes de voyageurs de commerce présentent une valeur de $\nu_{\mathrm{mdd}}$ inférieure à 27 389.

Les contraintes *sequence* (ici avec *cardinalité*) que l'on peut rencontrer dans les problèmes de *Car Sequencing* sont de forme pseudo-booléenne et peuvent être facilement modélisées sous forme de BDD (MDD à nœuds binaires) (Eén & Sörensson, 2006). Les contraintes sont de très forte arité : jusque 400 sur les données de la *CSPLib* de Gent et Walsh (1999). Il n'est pas envisageable de les résoudre avec des tables mais leur représentation sous forme de BDD est très compacte. Les problèmes de *Knapsack* sont formés de deux grandes contraintes linéaires ($\sum_{i=1}^{k} c_i x_i \leq c_0$) et ont un comportement analogue.

Les *nonogrammes* sont modélisés par des automates (contrainte *regular*) qui sont aisément « déroulés » en MDD. Ici, le taux de remplissage est suffisamment faible pour utiliser STR, mais MDDF est clairement plus efficace.

Les problèmes de voyageur de commerce (*TSP*) ou de mots-croisés (*CrossWord*) illustrent une limite de *MDDF* (et *MDDC*) par rapport à *STR2* : quand le MDD ne compresse pas efficacement la liste des $k$-uplets (on a ici $\nu_{\mathrm{mdd}} > \lambda$), *STR2* se révèle la meilleure implémentation de l'algorithme de filtrage. Les performances de *MDDF* restent cependant acceptables.

La faible consommation mémoire de *STR2* sur les problèmes de mots-croisés s'explique par le fait que toutes les tables de même arité sont identiques sur ces problèmes, ce qui permet de partager les $k$-uplets. *MDDF* ne réalise pas ce partage pour permettre une optimisation (cf fin de la section 6). Il devrait être possible d'améliorer l'implémentation des MDD sur ce point.

## 9. Conclusion

Dans cet article, nous avons présenté *MDDF*, un algorithme de révision de contraintes définies sous forme de MDD utilisant une file de propagation à gros grain. Cette structure permet de modéliser efficacement des contraintes définies sous forme d'une liste exhaustive de $k$-uplets autorisés, ainsi que certaines contraintes structurées (notamment les contraintes linéaires comme *sequence*). Ce nouvel algorithme est plus rapide en pratique que l'algorithme *MDDC* (Cheng & Yap, 2010), au prix d'un surcout raisonnable en termes d'espace. Sur certains types de contraintes sans structure et à taux de remplissage faible, les MDD ne compressent pas efficacement les tables. Les performances de *MDDF* restent cependant comparables à *STR2* et il est possible de chercher à améliorer la compression en réordonnant les variables, éventuellement

dynamiquement, voire même de manipuler l'algorithme de recherche pour orienter la résolution vers des zones de l'espace de recherche où les MDD seront de taille réduite.

Nous avons également montré qu'il est possible d'implanter des algorithmes de révision efficaces en n'utilisant que des structures de données immuables. Ces structures de données facilitent considérablement l'implantation d'algorithmes parallélisés. Il s'agit d'une perpective intéressante de ce travail.

*Remerciements*

*Ces travaux ont été en partie financés par le Ministère de l'Éducation nationale, de l'Enseignement supérieur et de la Recherche, la région Nord-Pas-de-Calais, le CNRS et le Campus International sur la Sécurité et l'Intermodalité dans les Transports (CISIT).*

## Références

Bessière, C. & Régin, J.-C. (1997). Arc Consistency for General Constraint Networks : Preliminary Results. In *Proc. IJCAI'97* (p. 398–404).

Bollig, B. & Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, *45*(9), 993–1002.

Boussemart, F., Hemery, F. & Lecoutre, C. (2004). Revision ordering heuristics for the CSP. In *Proc. CPAI'04 workshop held with CP'04* (p. 29–43). Toronto, Canada.

Briggs, P. & Torczon, L. (1993). An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, *2*(1–4), 59–69.

Cheng, K. & Yap, R. (2010). An MDD-based GAC algorithm for positive and negative table constraints and some global constraints. *Constraints*, *15*(2), 265–304.

Driscoll, J., Sarnak, N., Sleator, D. & Tarjan, R. (1989). Making data structures persistent. *J. of Computer and System Science*, *38*, 86–124.

Eén, N. & Sörensson, N. (2006). Translating pseudo-boolean constraints into SAT. *JSAT*, *2*, 1–26.

Fredkin, E. (1960). Trie memory. *Comm. ACM*, *3*(9), 490–499.

Gange, G., Stuckey, P. J. & Szymanek, R. (2011). MDD propagators with explanation. *Constraints*, *16*(4), 407–429.

Gent, I., Jefferson, C., Miguel, I. & Nightingale, P. (2007). Data structures for generalised arc consistency for extensional constraints. In *Proc. AAAI'2007* (p. 191–197).

Gent, I. & Walsh, T. (1999). *CSPLib : a benchmark library for constraints*. Technical report APES-09-1999. Available from http ://csplib.cs.strath.ac.uk/.

Katsirelos, G. & Walsh, T. (2007). A Compression Algorithm for Large Arity Extensional Constraints. In *Proc. CP'2007* (p. 379–393).

Lecoutre, C. (2011). STR2 : optimized simple tabular reduction for table constraints. *Constraints*, *16*(4), 341–371.

Lecoutre, C. & Hemery, F. (2007). A Study of Residual Supports in Arc Consistency. In *Proceedings of IJCAI'2007* (p. 125–130).

Lecoutre, C., Likitvivatanavong, C. & Yap, R. (2012). A path-optimal GAC algorithm for table constraints. In *Proc. ECAI'2012* (p. 510–515).

Lecoutre, C. & Szymanek, R. (2006). GAC for Positive Table Constraints. In *Proc. CP'06* (p. 284–298). Nantes, France.

Lhomme, O. & Régin, J.-C. (2005). A fast arc consistency algorithm for n-ary constraints. In *Proc. AAAI'2005* (p. 405–410).

Mairy, J.-B., van Hentenryck, P. & Deville, Y. (2014). Optimal and efficient filtering algorithms for table constraints. *Constraints*, *19*(1), 77–120.

Odersky, M. et al. (2001–2014). The Scala Programming Language. http://www.scala-lang.org/.

Sabin, D. & Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of cp'94* (p. 10–20).

Ullmann, J. (2007). Partition search for non-binary constraint satisfaction. *Information Science*, *177*, 3639–3678.

van Dongen, M., Lecoutre, C. & Roussel, O. (2008). Third International CSP Solvers Competition. http://www.cril.univ-artois.fr/CPAI08.

van Hentenryck, P., Deville, Y. & Teng, C. (1992). A Generic AC Algorithm and its Specializations. *Artificial Intelligence*, *57*, 291–321.

Vion, J. (2006–2014). Concrete : a CSP solving API for the JVM. http://github.com/concrete-cp.

Xu, K., Boussemart, F., Hemery, F. & Lecoutre, C. (2007). A simple model to generate hard satisfiable instances. *Artificial Intelligence*, *171*, 514–534.