Consistance d'arc par MDD-réduction

Julien Vion

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

Résumé

Dans cet article, nous présentons *MDDR*, 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, *MDDR* parvient à une meilleure incrémentalité que *MDDC* proposé par Cheng & Yap en 2010 [2]. 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, *MDDR* est presque aussi performant que l'état de l'art *STR2* de Ullmann et Lecoutre [12, 7] sur les contraintes *table* non structurées, définies sous forme de liste d'affectations autorisées.

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 relative clarté théorique et sa souplesse pour la modélisation des problèmes de décision combinatoires. Ceux-ci peuvent très naturellement être décomposés en variables et contraintes : le problème consiste alors à déterminer s'il existe une affectation des variables qui satisfasse toutes les contraintes. La résolution des CSP discrets se fait généralement par exploration arborescente systématique, et filtrage par consistance d'arc (AC pour *Arc Consistency*) à chaque nœud. La consistance d'arc consiste simplement à supprimer toutes les valeurs dont on peut montrer qu'il n'existe pas de solution faisant apparaitre cette valeur et satisfaisant une contrainte donnée. Le filtrage par consistance d'arc est NP-difficile dans le cas général, mais il existe de nombreux cas particuliers de contraintes qu'il est possible de filtrer en temps et en espace polynomiaux (les plus évidentes étant les contraintes portant sur un nombre borné de variables).

La contrainte *table* bénéficie d'un intérêt particulier : il s'agit d'une contrainte définie par une liste exhaustive de k-uplets (une *relation*) qui vont satisfaire (on parle de k-uplets *autorisés*), ou au contraire invalider (on parle de *nogoods*) la contrainte (dite respectivement *positive* ou *négative*). Ce type de contrainte trouve sa place dans la modélisation de nombreux problèmes de configuration ou de bases de données. De plus, des représentations compressées de ces tables permettent de modéliser efficacement certaines contraintes structurées [2].

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

$$\{ \quad \langle X = b, Y = c, Z = b \rangle , \quad \langle X = a, Y = b, Z = a \rangle , \\ \langle X = a, Y = a, Z = a \rangle , \quad \langle X = c, Y = a, Z = c \rangle , \\ \langle X = c, Y = a, Z = a \rangle , \quad \langle X = a, Y = a, Z = c \rangle \}$$

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* (i.e., dont toutes les valeurs sont présentes dans les domaines des contraintes) où la valeur apparait. Dans l'exemple 1, si le domaine de chaque variable est égal à $\{a, b, c\}$, la valeur b du domaine de la variable 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.

Il existe deux stratégies principales pour trouver un support à chaque valeur [9] : soit on itère sur les k-uplets *valides* (c'est-à-dire appartenant au produit cartésien du domaine des variables), en vérifiant à chaque itération si le k-uplet est autorisé par la contrainte, soit on itère sur les k-uplets *autorisés* par la contrainte,

en vérifiant à chaque itération si le k-uplet est valide. La première approche est généralement retenue pour les contraintes lâches et/ou d'arité faible, mais elle ne peut être efficacement appliquée à des contraintes très fortes : pour une contrainte autorisant λ k-uplets sur un alphabet de d valeurs, sa complexité temporelle dans le pire des cas est alors en $O(k(d^k - \lambda))$. Dans ce cas, on préfère la seconde option – en $O(k\lambda)$. Des algorithmes relativement efficaces ont été proposés pour cette approche : GAC-schema combiné à une représentation indexée ou compressée de la table de k-uplets [5, 10, 6] ; ou encore des algorithmes spécifiques comme *STR* [12, 7, 8], *AC5TC* [11] ou *MDDC* [2].

La représentation des tables par un diagramme de décision multi-valué (MDD) est l'une des approches les plus prometteuses, d'une part en termes d'espace (les MDD peuvent représenter un nombre exponentiel de k-uplets en espace polynomial), mais aussi de temps et de modélisation [2]. Dans cet article, nous proposons *MDDR*, un nouvel algorithme permettant d'établir la consistance d'arc sur une telle contrainte.

2 Définitions et notations

Définition 1 (Réseau de contraintes, variable, domaine, contrainte, affectation)**.** Un *réseau de contraintes* $\mathcal{N}$ est un couple $(\mathscr{X}, \mathscr{C})$ constitué :

- d'un ensemble de $\boldsymbol{n}$ *variables* $\mathscr{X}$; un *domaine* $\mathrm{dom}(X)$ est associé à chaque variable $X \in \mathscr{X}$ et définit l'ensemble fini d'au plus $\boldsymbol{d}$ valeurs auxquelles X peut être affectée, et
- d'un ensemble de $\boldsymbol{e}$ *contraintes* $\mathscr{C}$; chaque contrainte $C \in \mathscr{C}$ implique au plus $\boldsymbol{k}$ variables $\mathrm{vars}(C) \subseteq \mathscr{X}$; la contrainte définit au plus $\boldsymbol{\lambda}$ *affectations* autorisées $\mathrm{rel}(C)$ de ces variables.

La lâcheté d'une contrainte $\boldsymbol{l}$ (*looseness* ou *taux de remplissage*), comprise entre 0 et 100 %, est telle que $\lambda = ld^k$. Sa dureté t (*tightness*) est alors telle que $t + l = 100\,\%$.

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

Définition 2 (Consistance d'arc, support)**.** Soit C une contrainte et $X \in \mathrm{vars}(C)$. La valeur $v \in \mathrm{dom}(X)$ est arc-consistante (AC) pour C ssi il existe une affectation de $\mathrm{vars}(C)$, autorisée par C, qui instancie X à v (une telle affectation est appelée 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.

Fonction AC($\mathcal{N} = (\mathscr{X}, \mathscr{C})$, δ)

```
1  (Q, modif) ← initQueue(δ)
2  tant que Q ≠ ∅ faire
3      Choisir et retirer C de 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 − C | vars(C') ∩ Δ ≠ ∅}
11 retourner 𝒩
```

Notre définition de la consistance d'arc est généralisée aux contraintes d'arité quelconque. 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 CN 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 *relation*, c'est-à-dire 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^{e} élément du tableau. Pour une séquence $S = \langle 1, 2, 3\rangle$, on définit l'opérateur d'ajout :: tel que $0 :: S = \langle 0, 1, 2, 3\rangle$.

3 Algorithmes de base

On considère que le CSP est résolu par une recherche arborescente systématique, en profondeur d'abord, avec maintien de la consistance d'arc à chaque nœud (MAC). La fonction `AC` établit la consistance d'arc à l'aide d'une file de propagation à gros grain, contenant l'ensemble des contraintes à réviser, par un appel à la fonction `revise` qui lui est rattachée (ligne 4). Celle-ci renvoie la liste des variables ainsi modifiées (ou la valeur spéciale $\perp$ si une inconsistance, i.e., un domaine ou une relation vide, a été détectée), et 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

Fonction seekSupports(C)

```
1  ∀X ∈ vars(C), supp(X) ← ∅
2  seek ← vars(C)
3  pour chaque τ ∈ rel(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
```

maintient pour chaque contrainte la liste *modif* des variables modifiées depuis la dernière révision de celle-ci (lignes 7 à 9). La fonction `initQueue` initialise la file de propagation en fonction des décisions logiques utilisées pour la recherche systématique (typiquement, une valeur affectée à une variable), d'une manière similaire aux lignes 7 à 10 de l'algorithme.

Un algorithme simple pour réviser une contrainte définie comme une liste de k-uplets autorisés consiste à appliquer la fonction `seekSupports`. La fonction renvoie la liste des valeurs ayant un support, on peut alors supprimer toutes les autres valeurs des domaines des variables. L'algorithme parcourt la liste des k-uplets autorisés (ligne 3) et vérifie leur validité (ligne 4) : pour chaque k-uplet τ, $\tau[X]$ est la valeur du k-uplet correspondant à la variable X. Enfin, on complète la liste des valeurs supportées (lignes 5 et 6). Si toutes les valeurs de toutes les variables sont déjà supportées, on peut interrompre l'algorithme (lignes 7 à 9).

La fonction `seekSupports` admet une complexité temporelle en $O(k\lambda)$. Elle peut être appelée par `AC` jusqu'à kd fois par contrainte tout au long d'une branche de l'arbre de recherche : à chaque fois qu'une valeur est supprimée du domaine d'une variable impliquée par la contrainte. Cette version ne présente aucune propriété d'incrémentalité : la fonction `AC` admet alors une complexité de $O(ek^2d\lambda)$ pour une branche de l'arbre de recherche.

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

Initialement proposé en 2007 par J.R. Ullmann [12], puis amélioré par C. Lecoutre [7] (sous le nom *STR2*), l'algorithme *STR* améliore la fonction `seekSupports` en supprimant préalablement de rel(C) tous les k-uplets non valides : cela revient à effectuer l'opération de la ligne 4 préalablement à rechercher les supports de chaque valeur, et à maintenir cette version filtrée

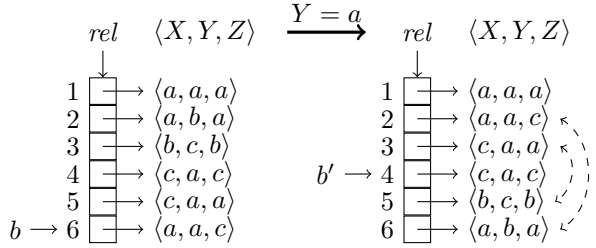


FIGURE 1 – Filtrage d'une relation (rel, b) par *STR*

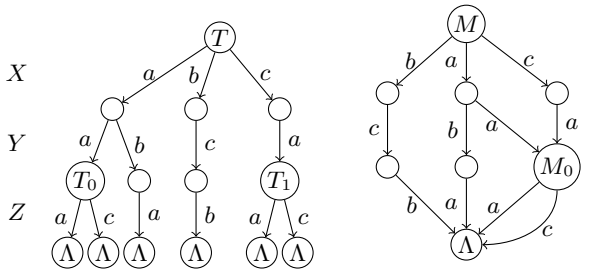
Fonction filterSTR((rel, b), *mod*)

```
1  b′ ← b ; i ← 1
2  tant que i ≤ b′ faire
3      si ∀X ∈ mod, rel[i][X] ∈ dom(X) alors
4          i ← i + 1
5      sinon
6          permuter rel[i] et rel[b′]
7          b′ ← b′ − 1
8  retourner (rel, b′)
```

de la relation entre deux appels à l'algorithme. Cette stratégie permet de ne contrôler la validé des k-uplets qu'au niveau des variables modifiées (c'est le principal apport de *STR2*). De plus, un k-uplet détecté invalide une fois ne sera plus considéré dans la même branche de l'arbre de recherche.

Cependant, maintenir une version filtrée de rel(C) au cours de la recherche n'est pas trivial : il faut être capable de restaurer rel(C) à une version antérieure lorsque l'on change de branche dans l'arbre de recherche (typiquement, après un retour-arrière). *STR* propose une astuce d'implantation permettant de sauvegarder et restaurer une liste de k-uplets en temps constant tout en n'empilant qu'un simple nombre entier à chaque niveau de l'arbre de recherche. Cf figure 1 et fonction `filterSTR` : *mod* contient la liste des variables modifiées depuis le dernier appel à la fonction. La relation est représentée par un couple (rel, b) : *rel* est un tableau de k-uplets. b est un nombre entier représentant l'index du dernier k-uplet valide (initialement, $b = \lambda$). Ainsi, les lignes 3 et 4 de `filterSTR` remplacent la ligne 4 de `seekSupports`. Pour supprimer un k-uplet de la structure, on le permute avec $rel[b]$, puis on décrémente b (lignes 7 et 8 de la fonction `filterSTR`). Il suffit de sauvegarder b pour être capable de restaurer la structure à un état antérieur.

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.



(a) Trie (b) MDD

FIGURE 2 – Un Trie T et un MDD M représentant la relation de l'exemple 1. T_0 et T_1 sont identiques : ils sont fusionnés en un seul nœud M_0 dans le MDD correspondant.

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 relation par STR2 tout au long d'une branche d'un arbre de recherche est en $O(kd\lambda)$ (i.e., un facteur k par rapport à STR). Cependant, une telle incrémentalité théorique n'a pas été observée pour la fonction `seekSupports` (bien qu'en pratique on constate que le temps passé à filtrer est généralement bien plus important que le temps passé à chercher les supports, grâce notamment aux optimisations apportées par la structure *supp*). La complexité globale d'`AC` reste donc inchangée.

La dernière amélioration de l'algorithme, *STR3* [8], atteint l'optimal $O(e(k\lambda + m))$ le long d'une branche de l'arbre de recherche de profondeur m [1] 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 que les précédentes versions de l'algorithme et nécessite l'utilisation d'une file de propagation à grain fin (i.e., maintenant la liste des valeurs supprimées entre deux révisions). Enfin, Mairy *et al.* ont récemment proposé une famille d'algorithmes, dont certains théoriquement optimaux, pour la propagation de contraintes *table* : *AC5TC* [11]. 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 certains types de contraintes très structurées (e.g., sous forme de MDD).

5 Tries et MDD

Le Trie (pour *Re*trie*val*) est une structure de données arborescente utilisée pour représenter un ensemble R de k-uplets de taille variable (le plus souvent des chaines de caractères) en un espace limité, et vérifier l'appartenance d'une chaine en $O(k)$ – avec un comportement théorique meilleur qu'une table de hash. Pour notre application (tous les k-uplets sont de même taille non-nulle), on peut le définir récursivement comme une application $T : i \to T'$ telle que T' est un Trie qui représente l'ensemble des $(k-1)$-uplets $\{\tau, \ldots\}$ tels que $(i :: \tau) \in R$. T' peut être une feuille (notée Λ), c'est-à-dire représenter l'ensemble $\{\langle\rangle\}$.

T forme un arbre de ν_{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) : le Trie réalise une *compression* de la relation. Chaque feuille du Trie représente un k-uplet de la relation d'origine. Dans le pire des cas, aucun préfixe commun ne peut être identifié (i.e., toutes les premières valeurs de chaque k-uplet sont différentes), et le Trie contient alors $k\lambda$ nœuds (notez que cela nécessite $d \geq \lambda$). On peut également montrer que $\nu_{\text{trie}} \in O(d^k)$, ce qui peut être intéressant – par rapport à une table en $O(kd^k)$ – si d et $k \ll \lambda$ (ce qui implique généralement un taux de remplissage l important).

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 de ν_{mdd} arcs (cf figure 2b). Pour une même relation, on a donc $\nu_{\text{mdd}} \leq \nu_{\text{trie}} \leq k\lambda$. Pour construire le MDD à partir d'une relation 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 celui-ci (fonction `mddReduce` de [2]). L'opération totale se fait en $O(k\lambda + d\nu_{\text{trie}})$. Le MDD d'une relation structurée peut souvent être construit directement par une fonction ad-hoc, mais le principe de l'indexation reste souvent présent. On atteint alors généralement une complexité en $O(\nu_{\text{trie}})$.

Différents ordonnancements des variables peuvent conduire à des Tries et MDD de taille différente. Trouver un ordonnancement de variables conduisant au Trie ou MDD de taille minimale est un problème NP-difficile [1] qui ne sera pas traité dans cet article.

L'utilisation de Tries et MDD pour représenter des contraintes *table* a déjà été envisagée, notamment dans les travaux de Gent *et al.* en 2007 (pour les Tries [5]) et de Cheng & Yap en 2010 (pour les MDD [2]). L'algorithme de Gent *et al.* recherche un support pour chaque valeur en parcourant successivement le Trie. Par exemple, on peut rechercher un support pour $X = b$ alors que $Y = a$ dans le Trie T de la figure 2a : partant de la racine T, on suit l'arc b (puisque $X = b$).

1. On remarquera que $m \leq nd$ dans la mesure où une hypothèse va supprimer au moins une valeur pour être exploitable.

L'arc suivant est invalide, puisque $Y \neq c$. Il n'y a pas d'autre arc disponible à ce niveau : $X = b$ n'a pas de support et peut être supprimée. Rechercher un support pour une valeur d'une variable située en bas de l'arbre nécessite de parcourir beaucoup plus de nœuds. Gent *et al.* proposent de générer plusieurs copies du même Trie : une pour chaque variable, le Trie étant réordonné pour que cette variable soit en haut de l'arbre. Cela ne tient malheureusement pas compte du fait qu'il est possible qu'une variable située en bas de l'arbre ait aussi pu être réduite à une seule valeur. Par exemple, si $Z = b$, il faut parcourir tout le sous-arbre de gauche pour montrer que $X = a$ n'a pas de support.

En plus d'utiliser des MDD, dont le nombre de nœuds est moindre que les Tries (et sont donc plus économes en mémoire et en temps), Cheng & Yap proposent dans leur algorithme *MDDC* un système permettant de sauvegarder, d'une exécution à l'autre de l'algorithme, quelles sous-parties du DAG ont été démontrées valides ou invalides. Ainsi, lorsque l'on rencontre plusieurs fois le même nœud au cours de la recherche, ou même au cours de deux recherches successives, on peut récupérer le résultat précédent immédiatement. 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* enregistre les nœuds invalides d'une exécution à l'autre de l'algorithme. Les *sparse sets* peuvent être « sauvegardés » et « restaurés » en temps constant (la technique est similaire à *STR*) en cas de retour-arrière.

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. Notre algorithme développe plus efficacement les possibilités d'incrémentalité exploitables sur une telle structure.

6 MDDR (MDD-Reduction)

Notre proposition s'inspire des structures de données immuables (*immutables*) utilisées dans les langages de programmation fonctionnels comme ML, Haskell ou Scala. Ces langages découragent ou interdisent la modification des données au cours de l'exécution des programmes : quand on ajoute un élément à une structure de données, on en crée une nouvelle, « 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. De ce fait, la complexité temporelle des opérations sur les structures immuables est généralement la même

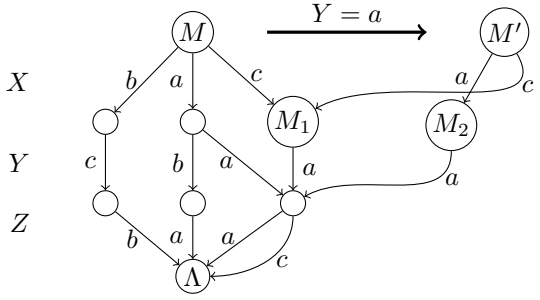


FIGURE 3 – Filtrage d'un MDD immuable.

que sur des structures modifiables. Les structures immuables simplifient considérablement la conception et le débogage des algorithmes, mais elles permettent surtout de s'affranchir de toute gestion de la synchronisation lors de la parallélisation.

Une version légèrement dégradée de *STR(2)* peut être implantée en utilisant des structures de données immuables. Il suffit de représenter la relation sous la forme d'une liste simplement chainée de k-uplets. Lorsque l'on filtre la relation, 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 (on ajoute cependant l'allocation d'une cellule de liste chainée à chaque fois que l'on vérifie la validité d'un k-uplet, et la libération de la mémoire en cas de retour-arrière – transparente pour le développeur si un ramasse-miettes est utilisé). En terme d'espace, on ajoute une liste chainée de taille $O(\lambda)$ à chaque niveau de l'arbre de recherche (il n'est pas nécessaire de dupliquer les k-uplets eux-mêmes, qui sont déjà immuables). Coder la phase de filtrage de *STR(2)* peut ainsi se faire en une seule ligne dans un langage fonctionnel.

L'idée centrale de notre contribution est d'appliquer un principe similaire à cette version immuable de *STR* aux MDD : la phase de filtrage au cœur du principe de *STR* est appliquée à un MDD. Une fonction (`filterMDD`) renvoie une « copie » filtrée du MDD, tout en conservant telles quelles les parties non modifiées du MDD, comme illustré sur la figure 3 : 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 et générer un seul nouveau nœud. Le gain potentiel par rapport à *STR* est de $O(\lambda)$ opérations ; et ce sans aucun surcout théorique en termes de complexité temporelle (avec cependant une restriction par rapport à *STR2*, discu-

Fonction filterMDD($M, ts, mod, \mathcal{X}$)

```
1  si M est vide alors retourner M
2  si mod = ⟨⟩ alors retourner M
3  si ts ≠ Ts(M) alors
4      Ts(M) ← ts
5      M' ← nouveau tableau de d MDD vides
6      X_m :: mod' ← mod
7      X :: X' ← X
8      si X_m = X alors
           /* Var modifiée au niveau courant */
9          pour chaque M[i] non vide | i ∈ dom(X)
           faire
10             M'[i] ← filterMDD(M[i], ts, mod', X')
11     sinon
12         pour chaque M[i] non vide faire
13             M'[i] ← filterMDD(M[i], ts, mod, X')
14     Last(M) ← si M = M' alors M sinon M'
15 retourner Last(M)
```

Fonction seekSuppMDD(M, ts, $supp$, $\mathcal{X}$, $seek$)

```
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)
```

tée ci-après). Le fait que le MDD d'origine ne soit pas modifié permet de garder une trace de celui-ci et de réaliser un retour-arrière en temps constant.

Dans un premier temps, nous avons implanté un Trie strictement immuable en Scala. Un Trie est représenté par un tableau de d éléments, chacun contenant le sous-Trie (éventuellement vide) préfixé par la valeur de l'arc correspondant à l'index du tableau. Une fois le Trie construit, il est réduit en MDD à l'aide de la fonction `mddReduce` de Cheng & Yap [2] en $O(d\nu_{\text{trie}})$.

Le cœur de notre contribution est la fonction `filterMDD` ci-dessus. Elle prend en paramètre un MDD M, un *timestamp* ts, i.e., un nombre entier différent à chaque appel à l'algorithme, une séquence mod contenant la liste des variables modifiées depuis le dernier appel à l'algorithme (générée par la fonction `AC`), et la séquence $\mathcal{X}$ des variables impliquées par la contrainte, dans l'ordre qui a servi à générer le MDD. Les variables doivent apparaitre dans le même ordre dans mod et $\mathcal{X}$.

Les deux premières conditions d'arrêt de la fonction récursive sont triviales : le MDD est inchangé s'il est vide ou si aucune variable n'a été modifiée (lignes 1 et 2). Le *timestamp* sert à ne parcourir chaque nœud qu'une seule fois par appel à l'algorithme. 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 3, 4, 14 et 15).

On récupère le premier élément X_m et X de chacune des séquences mod et $\mathcal{X}$ (lignes 5 et 6) : si $X_m = X$, c'est que la variable correspondant au niveau courant du MDD a été modifiée : il faudra contrôler la validité de chacune des valeurs au niveau courant. Les appels récursifs se feront avec mod' (i.e., mod amputé de cette variable). Une fois toutes les variables modifiées traitées, il devient inutile de parcourir les nœuds situés plus bas dans le graphe (test de la ligne 2). Notez que ce test permet aussi de contrôler si l'on est arrivé en bas du graphe (i.e., sur une feuille) : à ce moment, toutes les variables ont forcément été traitées.

À chaque fois, on reconstitue un nouveau nœud M' (lignes 5 et boucle des lignes 9-10, ou 12-13) créé pour l'occasion. Si le nouveau nœud M' est identique à l'ancien M, il peut être libéré et M utilisé à la place. Il est possible qu'après un appel à la fonction `filterMDD`, 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 chercher à réduire les MDD après chaque filtrage soit trop couteux, ce qui a été confirmé par des tests préliminaires.

La fonction `seekSuppMDD` 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. Les paramètres sont le MDD M, un *timestamp* ts (différent du *timestamp* de `filterMDD`), l'ensemble des supports trouvés $supp$ (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 des valeurs n'ont pas de support trouvé $supp$ (initialement, toutes les variables impliquées par la contrainte). Ici encore, on utilise les *timestamps* pour ne pas traiter deux fois le même nœud (lignes 1 et 2) ; l'ensemble $seek$ permet d'arrêter le parcours si un support pour chaque va-

leur des variables situées plus bas dans le MDD a déjà été trouvé (similaire au *early cutoff optimization* de *MDDC*, mais à grain plus fin, lignes 5, 7 et 8). À la fin de l'exécution de l'algorithme, l'ensemble *seek* ne contient plus que les variables pour lesquelles des valeurs n'ont pas de support : seules ces variables doivent être filtrées.

Pour optimiser l'exécution des boucles des lignes 9 et 12 de `filterMDD` ou de la ligne 4 de `seekSuppMDD`, 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, par exemple si plusieurs contraintes partagent la même relation ; l'implantation n'est plus non plus strictement immuable. 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 seul fils ou à deux fils (i.e., nœuds unaires et binaires).

7 Complexités et discussion

Observation 1. *La complexité temporelle dans le pire des cas de* `filterMDD` *est en* $O(\nu_{\text{mdd}})$.

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 $O(\nu_{\text{mdd}})$. 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 (c'est ce que fait la boucle de la ligne 12). Les propriétés d'incrémentalité de *MDDR* 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.

Observation 2. *La complexité temporelle dans le pire des cas de* `seekSuppMDD` *est en* $O(\nu_{\text{mdd}})$.

Le test à la ligne 5 de la fonction `seekSuppMDD` peut être réalisé en temps amorti $O(k+\nu_{\text{mdd}})$ 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, et on a $k \leq \nu_{\text{mdd}}$ si $\lambda > 0$ ($\lambda = 0$ implique un CN trivialement inconsistant), d'où l'observation.

Observation 3. *La complexité temporelle dans le pire des cas de* `AC` *tout au long d'une branche de l'arbre de recherche si toutes les contraintes sont filtrées par* MDDR *est en* $O(ekd\nu_{\text{mdd}})$.

Aucune propriété théorique d'incrémentalité n'a été observée sur `filterMDD` ni `seekSuppMDD`. Dans le pire des cas, chaque appel à l'algorithme concerne la modification de la variable située le plus bas dans le MDD, ce qui nécessite de parcourir tous les arcs du graphe. Cependant, le nombre de *vérifications de domaine* effectuées par *MDDR* sera en $O(ed\nu_{\text{mdd}})$ tout au long d'une branche de l'arbre de recherche. Après avoir filtré la relation et recherché les supports, il faut également supprimer les valeurs n'ayant pas de supports. Il est évident que ce cout est $O(nd)$ amorti sur toute la branche de l'arbre de recherche, chaque valeur ne pouvant 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* est tout au long d'une branche de l'arbre de recherche de $O(ek^2d\lambda)$ ($O(ekd\lambda)$ pour les vérifications de domaine de la phase de filtrage), celle de *MDDC* est $O(ekd\nu_{\text{mdd}})$ (tant en termes d'opérations que de vérifications de domaine).

Observation 4. *Si toutes les contraintes sont filtrées par* MDDR*, la complexité spatiale dans le pire des cas de* `AC` *tout au long d'une branche de l'arbre de recherche est* $O(ekd\nu_{\text{mdd}})$.

Il est intéressant de noter que chaque arc supprimé ne nécessite de générer qu'un nœud à chaque niveau du MDD, soit $O(kd)$ arcs. Au total, au maximum $O(kd\nu_{\text{mdd}})$ arcs (par contrainte) peuvent ainsi être générés tout au long de la recherche. À titre de comparaison, les complexités spatiales de *STR2* et *MDDC* sont respectivement $O(ek\lambda)$ et $O(e\nu_{\text{mdd}})$.

L'algorithme *MDDC* de Cheng & Yap met en place une incrémentalité en enregistrant les nœuds valides et invalides : un nœud est valide si au moins un arc partant de ce nœud conduit à un nœud valide. L'information concernant les nœuds invalides peut être conservée entre deux appels successifs. *MDDR* va plus loin : en enregistrant une copie du MDD, *MDDR* est capable d'éliminer chaque *arc* non valide : l'incrémentalité obtenue est bien meilleure, et ce sans surcout en termes de complexité temporelle. De plus, *MDDR* permet d'exploiter plus finement les informations sur les variables modifiées fournies par l'algorithme de propagation `AC` (i.e., structure *mod* de `filterMDD`, à la manière de l'optimisation effectuée entre *STR* et *STR2*). *MDDC* tel que décrit par Cheng & Yap n'exploite aucunement les informations concernant les variables modifiées. Il est possible de modifier l'algorithme pour

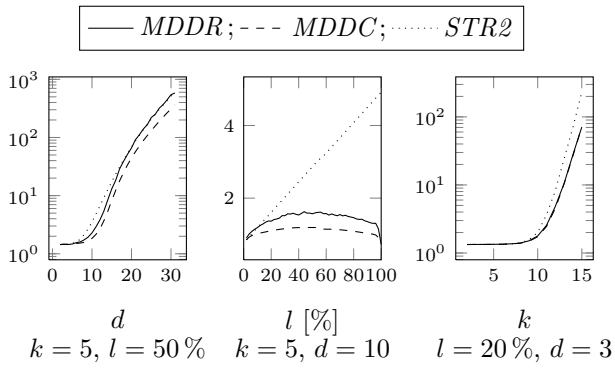


FIGURE 4 – Mémoire utilisée (en Mio) au 20e niveau d'un arbre de recherche simulé (médianes sur 50 exécutions).

les prendre en compte, mais les structures de données de *MDDC* ne permettent pas de le faire efficacement, puisque il faut vérifier à nouveau au moment de la recherche des supports si les arcs sont effectivement valides. Dans le cas où seule la variable tout en haut de l'arbre a été modifiée, *MDDR* permet de ne vérifier que $O(kd)$ valeurs alors que *MDDC* va potentiellement parcourir tout le graphe.

Finalement, *MDDR* 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 *MDDR* est presque toujours plus rapide que *MDDC*, il est surtout intéressant par rapport à *STR2* quand $\nu_{\text{mdd}} \ll k\lambda$, ce qui implique des contraintes structurées et/ou un taux de remplissage des relations (l) important.

8 Expérimentations

Les algorithmes ont été implantés en langage Scala 2.10 sur notre solveur CSP4J 2 (le code source est disponible sur le site du projet), exécuté 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, disposant de 8 GiB RAM. Les algorithmes de révision testés pour les contraintes d'arité supérieure à 2 définies en extension sont *STR2*, *MDDC* et *MDDR*. 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 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 dans [2]) d'après six paramètres (n, d, k, e, l, q) : respectivement le nombre de variables n, de valeurs d, l'arité des contraintes k, leur nombre e, leur taux de remplissage l (en termes de probabilité), et enfin un « taux de structure » q, qui définit la probabilité qu'a un nœud du MDD d'être identique à un autre. 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 l, ou supprimée. Enfin, chaque nœud est remplacé avec une probabilité q par un autre nœud déjà généré.

La figure 4 montre l'évolution de l'utilisation de la mémoire pour stocker une contrainte quand l'un des paramètres d, l ou k augmente (on a gardé $q = 0$ ici). Pour montrer l'impact de la création de nouveaux nœuds de MDD au cours de la recherche avec *MDDR*, une branche d'arbre de recherche de profondeur 20 est simulée en retirant au hasard 5 % des valeurs et en effectuant une révision à chaque nœud (courbes du bas sur la figure). On remarque qu'avec les MDD, la mémoire atteint rapidement un maximum (en $O(d^k)$ avec une constante relativement faible) quand l augmente, et décroit quand l approche les 100 % (une relation « complète » se représente en dk arcs avec un MDD), là où une représentation en table voit sa taille augmenter linéairement. Cette courbe laisse supposer que les algorithmes présentés restent performants même avec des taux de remplissage très importants (i.e., contraintes définies sous forme de k-uplets interdits). On constate que *MDDR* utilise plus de mémoire que *MDDC* au cours de la recherche, mais cela reste raisonnable : la mémoire utilisée reste inférieure à *STR2* avec ces paramètres. Enfin, la courbe la plus adroite laisse supposer que le paramètre k n'a en réalité pas d'impact sur la complexité spatiale de *MDDR*, et permet de conjecturer que sa complexité spatiale en moyenne est en $O(ed\nu_{\text{mdd}})$.

Nous avons souhaité observer l'évolution des performances en fonction du taux de remplissage (auquel le nombre de k-uplets λ ou le nombre de nœuds des MDD ν_{mdd} sont directement liés) sur un problème de base tel que $(d, k) = (5, 5)$, puis en augmentant séparément d et k à 8. Le nombre de variables n a été ajusté en conséquence pour que les problèmes restent de difficulté moyenne (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 [13]. Pour une meilleure

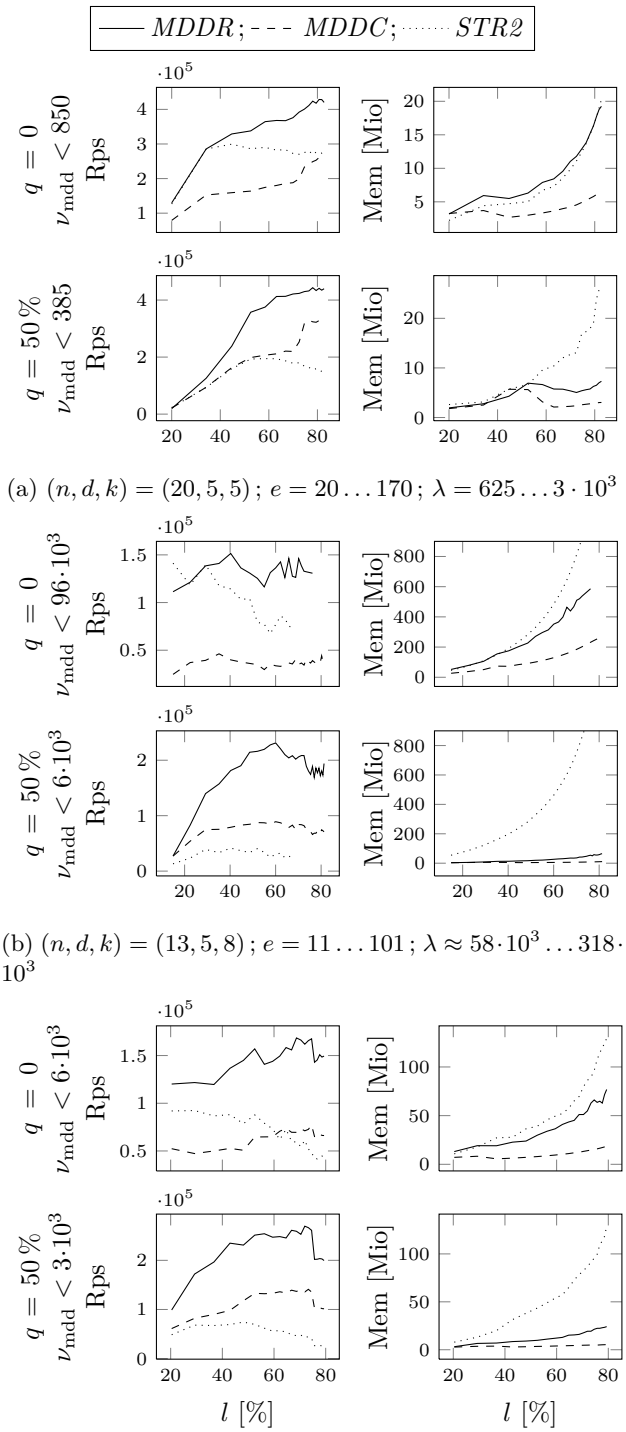
précision, nous avons fixé le nombre de contraintes et calculé le taux de remplissage : $l = \exp(-e^{-1} n \ln d)$. Rappelons que pour que ces théorèmes s'appliquent, il faut que $l \geq k^{-1}$, ce qui fixe la limite basse de nos tests. Enfin, nous avons testé des problèmes sans aucune structure ($q = 0$) ou au contraire assez structurés ($q = 50\,\%$). Des valeurs indicatives de e, λ et ν_{mdd} sont indiqués sur les graphes.

Les résultats sont présentés sur la figure 5. Rappelons que 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. Nous avons donc préféré évaluer la performance des algorithmes à partir du nombre de révisions par seconde plutôt que par le temps total de résolution : les courbes résultantes sont bien plus lisibles. Rappelons que l'arbre de recherche ainsi que la séquence de révisions sont strictement identiques entre les trois versions présentées.

On constate que sur ces problèmes aléatoires, *MDDR* est toujours plus rapide que *MDDC*. Il est aussi plus rapide que *STR2* à l'exception des problèmes ayant une arité forte et un taux de remplissage faible : c'est dans ces conditions que la limitation des possibilités d'incrémentalité de *MDDR* par rapport à *STR2* a le plus d'impact. Sur des problèmes structurés, *STR2* est bien sûr de très loin le moins efficace. *MDDC* reste toujours le moins gourmand en espace mémoire, grâce à ses structures de données minimalistes.

Des tests préliminaires sur des problèmes plus structurés confirment les bons résultats de *MDDR*. Les contraintes de *séquence* (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 (i.e., MDD à nœuds binaires) [3]. Les contraintes sont de très forte arité (jusque 400 sur les données de la *CSPLib* [4]) ; il n'est pas envisageable de les résoudre avec des tables, mais leur représentation sous forme de BDD est très compacte (cf table 1). Les problèmes de *Knapsack* sont formés de deux grandes contraintes linéaires ($\sum_{i=1}^{k} c_i x_i = 0$) et ont un comportement analogue.

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



(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 – Vitesse de propagation (révisions par seconde) et mémoire utilisée pour résoudre des problèmes aléatoires. Chaque point est une médiane de 25 problèmes générés et résolus.

					MDDR		*MDDC*		*STR2*	
	d	k	λ	ν_{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	**37**	**7**	31	**7**	24	15
crossword	26	15	36 192	39 557	10	20	6	8	**19**	**5**
tsp	10^3	3	26 872	27 389	1 002	16	211	**6**	**1 054**	7

TABLE 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).

9 Conclusion

Dans cet article, nous avons présenté *MDDR*, 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 (e.g., contraintes pseudo-booléennes). Ce nouvel algorithme est plus rapide en pratique que l'algorithme *MDDC* proposé par Cheng & Yap en 2010 [2], au prix d'un surcout raisonnable en termes d'espace. Sur certains types de contraintes (i.e., sans structure et à taux de remplissage faible), les MDD ne compressent pas efficacement les tables. Les performances de *MDDR* restent cependant comparables à *STR2*, et il est possible de chercher à améliorer la compression en réordonnant les variables.

Nous avons également montré qu'il était possible d'implémenter des algorithmes de révision efficaces en utilisant exclusivement 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.

Références

[1] B. BOLLIG et I. WEGENER. « Improving the variable ordering of OBDDs is NP-complete ». Dans : *IEEE Transactions on Computers* 45.9 (1996), p. 993–1002.

[2] K.C.K. CHENG et R.H.C. YAP. « An MDD-based GAC algorithm for positive and negative table constraints and some global constraints ». Dans : *Constraints* 15.2 (2010), p. 265–304.

[3] N. EÉN et N. SÖRENSSON. « Translating pseudo-boolean constraints into SAT ». Dans : *JSAT* 2 (2006), p. 1–26.

[4] I.P. GENT et al. *A problem library for constraints*. `http://csplib.org/`. 1996–2012.

[5] I.P. GENT et al. « Data structures for generalised arc consistency for extensional constraints ». Dans : *Proc. AAAI'2007*. 2007, p. 191–197.

[6] G. KATSIRELOS et T. WALSH. « A Compression Algorithm for Large Arity Extensional Constraints ». Dans : *Proc. CP'2007*. 2007, p. 379–393.

[7] C. LECOUTRE. « STR2 : Optimized Simple Tabular Reduction for Table Constraints ». Dans : *Constraints* 16.4 (2011), p. 341–371.

[8] C. LECOUTRE, C. LIKITVIVATANAVONG et R.H.C. YAP. « A path-optimal GAC algorithm for table constraints ». Dans : *Proc. ECAI'2012*. 2012, p. 510–515.

[9] C. LECOUTRE et R. SZYMANEK. « GAC for Positive Table Constraints ». Dans : *Proc. CP'06*. Nantes, France, 2006, p. 284–298.

[10] O. LHOMME et J.-C. RÉGIN. « A fast arc consistency algorithm for n-ary constraints ». Dans : *Proc. AAAI'2005*. 2005, p. 405–410.

[11] J.-B. MAIRY, P. van HENTENRYCK et Y. DEVILLE. « An Optimal Filtering Algorithm for Table Constraints ». Dans : *Proc. CP'2012*. 2012.

[12] J.R. ULLMANN. « Partition search for non-binary constraint satisfaction ». Dans : *Information Science* 177 (2007), p. 3639–3678.

[13] K. XU et al. « A simple model to generate hard satisfiable instances ». Dans : *Artificial Intelligence* 171 (2007), p. 514–534.