Filtrage tardif du BIBD : lorsque la procrastination paie

Yassine Attik **Jonathan Gaudreault** **Claude-Guy Quimper**

Consortium de recherche FORAC, Université Laval, Québec, Canada

Yassine.Attik@cirrelt.ca {Jonathan.Gaudreault,Claude-Guy.Quimper}@ift.ulaval.ca

Résumé

Lorsque nous développons une nouvelle contrainte pour un solveur de programmation par contraintes, il faut implémenter les algorithmes de filtrage qui se déclenchent durant la recherche lorsque surviennent différents événements (e.g. *une variable a été instanciée*, *une valeur a été retirée du domaine*, etc.). Dans le cadre de cette recherche, nous cherchons à savoir quel est le meilleur moment pour filtrer lorsque nous tentons de résoudre le problème du *Balanced Incomplete Block Design* (BIBD). Retarder la majorité du filtrage jusqu'au dernier moment, ce qui revient à dire qu'on filtre uniquement lorsque la variable vient d'être sélectionnée et est sur le point d'être instanciée, réduit le temps de calcul entre 20 % et 30 %. L'approche préserve le même espace de solution et le même niveau de filtrage. Les solveurs génériques n'offrent pas de manière explicite la possibilité de rattacher le filtrage à l'événement *la variable vient d'être sélectionnée*. Nos travaux indiquent cependant que cela pourrait être intéressant de l'inclure dans les solveurs génériques. Bien sûr, l'approche ne garantit pas le même niveau de filtrage pour tous les problèmes. À cet effet, nous présentons un contre-exemple.

Abstract

When developing a new constraint for a CP solver, one must implement filtering algorithms that trigger during the search according to different events (e.g. *a variable is instantiated*, *a value is removed from the domain*, etc.). In the context of this research, we studied what is the best moment to carry on filtering while solving the Balanced Incomplete Block Design (BIBD) problem. It showed that delaying most of the filtering until the last moment that is, filtering the domain of a variable only when the variable is selected and is about to be instantiated, reduces computation time by 20 % to 30 %. The approach preserved the same solution space and the same level of filtering. Current generic solvers do not provide a *variable is selected* event to attach filtering algorithms to. It could be interesting to add this option into generic solvers. Of course, our approach does not guarantee the same filtering level for all the problems, which we illustrate with a counter-example.

1 Introduction

Le problème du Balanced Incomplete Block Design (BIBD) possède plusieurs applications importantes telles que le design de plans d'expérience, la planification d'horaire de tournoi, le *coding theory* [6, 17, 18] et l'optimisation de portfolios financiers [2]. Il peut être résolu en utilisant différentes approches telles que la recherche locale [9], la programmation en nombres entiers [18] ou la programmation par contraintes (PPC) [8, 10].

Dans le contexte de cette recherche, nous étudions quel est le meilleur moment pour effectuer le filtrage lorsque nous résolvons un BIBD en utilisant les techniques de PPC. Lorsqu'on développe une nouvelle contrainte pour un solveur de PPC, il faut aussi implémenter les algorithmes de filtrage qui seront déclenchés durant la recherche lorsque différents événements vont survenir. Les événements les plus communs sont l'instanciation d'une variable, la modification d'une borne (borne supérieure ou inférieure) ou, plus généralement, le retrait d'une valeur [12, 15, 16].

Nous essayons une autre approche que nous appelons le filtrage tardif (*lazy filtering*). Nous retardons le filtrage du domaine d'une variable jusqu'au dernier moment, c'est-à-dire, lorsque la variable a été sélectionnée et qu'elle est sur le point d'être instanciée. Il apparaît que, pour les BIBDs, cette approche offre exactement le même niveau de filtrage tout en réduisant le temps de résolution de 20 % à 30 %. Bien sûr, le résultat ne peut pas être généralisé à tous les problèmes en PPC, ce que nous illustrons avec un contre-exemple qui n'est pas un BIBD.

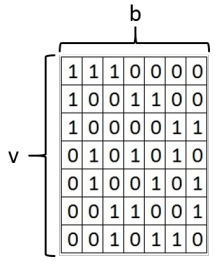

Image 1 – Représentation binaire

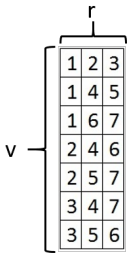

Image 2 – Représentation n-aire

Le reste de cet article est organisé ainsi. La section 2 présente les concepts préliminaires à propos des BIBDs, la PPC et le filtrage. La section 3 décrit comment les contraintes du BIBD peuvent être adaptées pour le filtrage tardif. La section 4 montre les résultats des expériences. La section 5 conclut l'article.

2 Concepts préliminaires

2.1 Balanced Incomplete Block Design

Un problème BIBD est défini par un tuple (v, b, r, k, λ). Nous avons v types d'objets qui doivent être assignés à b blocs de manière à ce que chaque bloc contienne k objets de différents types. Chaque type d'objets est présent dans r blocs différents. Pour n'importe quel pair de type d'objets, ils doivent apparaître simultanément dans λ différents blocs [1].

Par définition, les instances BIBDs doivent satisfaire les propriétés suivantes [17] :

1. $b = \frac{\lambda(v^2-v)}{k^2-k}$
2. $r = \frac{\lambda(v-1)}{k-1}$

De ce fait, une instance BIBD peut être définie en utilisant seulement les paramètres (v, k, λ).

Le problème peut être modélisé en utilisant une matrice de variables binaires ayant v lignes et b colonnes $X = (x_{ij}) \in \{0, 1\}^{v \times b}$, où x_{ij} est égal à 1 *ssi* un objet du type i est assigné au bloc j. Il peut aussi être modélisé en utilisant une matrice $Y = (y_{ij}) \in [1..b]^{v \times r}$ où pour chaque ligne, correspondant à un type d'objet i, nous indiquons dans quels r blocs de 1 à b ils sont placés. Ces deux représentations, mathématiquement équivalentes, sont illustrées par les images 1 et 2 qui montrent une même solution pour l'instance $(7, 3, 1)$.

Les trois premières lignes du Tableau 1 sont les contraintes requises pour représenter un BIBD sous la forme d'une matrice (versions binaire et n-aire). La représentation d'un BIBD sous forme matricielle amène de la symétrie qui est importante à considérer. En effet, pour n'importe quelle solution que nous trouvons, il existe $v! \times b!$ autres solutions symétriques [1, 4]. C'est pourquoi il est important d'utiliser des stratégies de bris de symétrie comme celles indiquées par les contraintes 4 et 5 du Tableau 1.

1. http ://www.csplib.org/Problems/prob028

2.2 Programmation par contraintes, filtrage et propagation

En PPC, chaque contrainte possède des algorithmes de filtrage dont le travail est de retirer les valeurs des domaines qui ne satisfont pas la contrainte.

Les algorithmes de filtrage sont déclenchés lorsqu'un domaine est modifié. Pour une contrainte donnée, un ou plusieurs algorithmes peuvent être associés à différents événements qui correspondent à différentes modifications qui sont survenues sur le domaine [16]. Les événements généralement reconnus par les solveurs sont : l'instanciation d'une variable, la modification d'une borne inférieure, la modification d'une borne supérieure et le retrait d'une valeur [12, 15, 16]. Schulte et Stuckey [15] proposent d'autres événements tels que lorsque toutes les valeurs d'un domaine deviennent positives ou lorsqu'une valeur spécifique est retirée du domaine. Les différents algorithmes de filtrage sont exécutés de manière répétitive (propagation) jusqu'au moment où aucun d'entre eux ne peut plus modifier de domaines davantage. Nous atteignons donc un *fixpoint* [12, 15, 16].

Il existe des situations particulières où 2 niveaux de filtrage deviennent équivalents [14]. Dans ces cas-là, il est préférable d'utiliser l'algorithme le plus rapide.

Avoir le même niveau de filtrage signifie que nous visitons les mêmes solutions ordonnées de la même manière tout en ayant un nombre d'échecs identique lors de l'exploration des arbres. Cela veut donc dire que nous avons exploré le même arbre de recherche pour une version ou une autre.

3 Filtrage tardif des BIBDs

Il existe généralement un compromis à faire entre le temps passé à explorer un arbre de recherche et le

Tableau 1 – Contraintes pour BIBD binaire vs n-aire. Les contraintes GCC et NValue sont définies dans [13] et [7] respectivement. La contrainte GCC est définie comme étant $GCC([X_1...X_n],[l_1...l_m],[u_1...u_m])$ où $\forall x \quad l_x \leq |\{i \mid X_i = x\}| \leq u_x$. La contrainte NValue est définie comme étant $NValue([X_1...X_n],x) \iff |\{X_1...X_n\}| = x$ GCC et NValue sont implémentés de manière simplifiée dans 3.2 ainsi que 3.3 considérant l'ordonnancement statique des variables que nous utilisons.

	Binaire	**n-aire**
1. Chaque type d'objet doit apparaître dans r blocs	$\sum_{j=1}^{b} x_{ij} = r \quad \forall i \in [1..v]$	$y_{ij} < y_{i,(j+1)} \quad \forall i \in [1..v]$ $\forall j \in [1..(r-1)]$ Cette contrainte brise partiellement la symétrie de colonnes.
2. Chaque bloc doit contenir exactement k objets distincts	$\sum_{i=1}^{v} x_{ij} = k \quad \forall j \in [1..b]$	$GCC(Y,[k_1,\ldots,k_b],[k_1,\ldots,k_b])$
3. Deux objets distincts doivent apparaître ensemble dans λ blocs	$\sum_{j=1}^{b} x_{ij}x_{lj} = \lambda$ $\forall i,l \in [1..v]$ *et* $i < l$	$NValue([y_{i1},\ldots,y_{ir},y_{l1},\ldots,y_{lr}],$ $2r-\lambda) \quad \forall 1 \leq l < i \leq v$
4. Ordre lexicographique sur les lignes	$x_{i,1},\ldots,x_{i,b} \prec_{\text{LEX}}$ $x_{(i+1),1},\ldots,x_{(i+1),b}$ $\forall i \in [1..(v-1)]$	$y_{i,1},\ldots,y_{i,r} \prec_{\text{LEX}} y_{(i+1),1},\ldots,y_{(i+1),r}$ $\forall i \in [1..(v-1)]$
5. Ordre lexicographique sur les colonnes	$x_{1,j},\ldots,x_{v,j} \succeq_{\text{LEX}}$ $x_{1,(j+1)},\ldots,x_{v,(j+1)}$ $\forall j \in [1..(b-1)]$	Nous pouvons facilement implémenter une heuristique de choix de valeur qui fait cela. Si nous avons le choix entre plusieurs valeurs différentes qui ont, jusqu'à maintenant, été utilisées sur les mêmes lignes, alors nous utilisons la plus petite valeur possible.

temps passé à filtrer cet arbre en utilisant les algorithmes de filtrage. Ce n'est pas le sujet de notre recherche présentée ici. Plutôt, nous montrons que, pour un BIBD, il est possible, en remettant le filtrage au dernier moment, de conserver le même niveau de filtrage tout en diminuant le temps de résolution.

Nous filtrons le domaine d'une variable au tout dernier moment, c'est-à-dire, entre le moment où la variable est sélectionnée et juste avant que l'heuristique de choix de valeur soit appelée. De ce fait, nous ajoutons un événement de filtrage au solveur qui s'appelle *variableAÉtéSélectionnée*. C'est une pratique plutôt inhabituelle puisque les solveurs préfèrent filtrer le plus tôt possible, c'est-à-dire, au moment où le domaine d'une autre variable est modifié.

Les sections 3.1 à 3.5 présentent comment les algorithmes de filtrages pour le BIBD peuvent être implémentés. Il est important de remarquer qu'un algorithme est différent selon que l'algorithme de filtrage se déclenche tardivement (*variableAÉtéSélectionnée*) ou avec les événements classiques (bornes de la variable modifiées ou variable instanciée). Cette différence est due au fait que, dans la version traditionnelle, nous pouvons modifier des domaines de n'importe quelle variable de la matrice, alors qu'en version tardive, nous modifions que le domaine de la variable qui vient d'être sélectionnée. Dans tous les cas, nous obtenons le même niveau de filtrage, mais avec des temps de résolution différents que nous allons évaluer dans la section 4.

Nous présentons les algorithmes et les résultats d'expériences pour la représentation n-aire des BIBD puisqu'ils étaient plus faciles à adapter avec notre technique. Nous donnons aussi l'analyse de la complexité des algorithmes. Des algorithmes équivalents existent pour le modèle binaire. L'implémentation des algorithmes de filtrage décrits suppose que les variables sont instanciées selon une séquence statique (de la gauche vers la droite, du haut vers le bas). Il s'agit d'une pratique courante pour la résolution du BIBD [2]. De même, nous instancions les variables en choisissant d'abord les plus petites valeurs (il s'agit cependant d'un choix arbitraire puisque, par définition, les valeurs ne sont pas réellement « numériques », mais

2. Choco [11] et Gecode [5] utilisent des heuristiques statiques pour la résolution du BIBD dans les fichiers exemples de leurs solveurs.

correspondent en réalité à des identifiants de blocs où on souhaite placer les objets).

3.1 Chaque type d'objet doit apparaître dans r blocs

Cette contrainte force les valeurs d'une ligne à prendre une valeur différente. On y arrive en imposant un ordre croissant. Cela permet aussi, partiellement, de briser une forme de symétrie.

3.1.1 Filtrage traditionnel

Dans une ligne i où la variable y_{ij} a été instanciée, nous filtrons le domaine de toutes les variables $l > j$ en utilisant la formule (1). La complexité de cet algorithme est $O(r)$.

$$y_{ij} < y_{il} \qquad l \in [(j+1)..r] \tag{1}$$

3.1.2 Filtrage tardif

Lorsque la variable y_{ij} est sélectionnée, nous avons seulement besoin de retirer du domaine courant les valeurs inférieures à la valeur de $y_{i,(j-1)}$ en utilisant la formule (2). La complexité de cet algorithme est $O(1)$.

$$y_{i,(j-1)} < y_{ij} \tag{2}$$

Avec cette version, chaque variable de la ligne est filtrée qu'une seule fois, alors qu'avec le filtrage traditionnel, les variables de la dernière colonne sont filtrées jusqu'à $r-1$ fois. Cependant, pour les deux algorithmes, le même nombre de valeurs sont retirées.

3.2 Chaque bloc doit contenir exactement k objets distincts

Cette contrainte requiert que chaque valeur apparaisse k fois dans la matrice. Dans la version traditionnelle, nous devons vérifier l'ensemble de la matrice lorsqu'une variable est instanciée. Dans la version tardive, nous avons seulement besoin de vérifier les variables précédemment instanciées.

3.2.1 Filtrage traditionnel

Lorsqu'une variable y_{ij} est instanciée, nous devons vérifier, pour toute la matrice, le nombre de fois que la valeur de y_{ij} est utilisée. Si la valeur a été utilisée k fois, alors nous devons retirer du domaine des variables qui suivent la valeur de y_{ij}. La complexité de cet algorithme est $O(vrb)$.

3.2.2 Filtrage tardif

Lorsque la variable y_{ij} est sélectionnée, nous devons vérifier, seulement pour les variables précédemment instanciées, combien de fois chaque valeur est instanciée. Par la suite, nous retirons du domaine de la variable y_{ij} les valeurs qui sont utilisées k fois. La complexité de cet algorithme, en pire cas, est $O(vr+b)$.

3.3 Deux objets distincts doivent simultanément apparaître dans λ blocs

Dans la version traditionnelle, il est nécessaire de vérifier les $v-1$ paires de lignes que la ligne courante peut former avec les autres lignes. Dans la version tardive, il est seulement nécessaire de vérifier les paires de lignes que la ligne courante peut former avec les lignes précédentes.

3.3.1 Filtrage traditionnel

À partir de la ligne i à laquelle la variable y_{ij} qui vient d'être instanciée appartient, nous devons considérer les $v-1$ paires de lignes formées avec les autres lignes.

Pour chaque paire, nous devons calculer, pour les variables déjà instanciées, le nombre de fois que chaque valeur est utilisée. Si pour une paire nous avons exactement λ valeurs qui sont utilisées deux fois, alors nous devons retirer de chaque ligne qui forment la paire les valeurs des variables instanciées de l'autre ligne. Si nous avons plus que λ valeurs utilisées deux fois, alors il faut forcer un échec. La complexité de cet algorithme est $O(vrb)$.

3.3.2 Filtrage tardif

Lorsque la variable y_{ij} est sélectionnée, nous devons, pour chaque $i-1$ paires qu'on peut former avec les lignes précédentes, calculer le nombre de fois que chaque valeur est utilisée. Si nous avons exactement λ valeurs utilisées deux fois, alors nous devons retirer du domaine de la variable sélectionnée y_{ij} les valeurs des variables instanciées de l'autre ligne de la paire. La complexité de cet algorithme est $O(r+b)$ dans le meilleur cas lorsqu'on filtre la première ligne avec la deuxième et le pire cas est $O(vr+b)$ lorsqu'on filtre la dernière ligne avec les autres au-dessus.

3.4 Ordre lexicographique entre les lignes

3.4.1 Filtrage traditionnel

Pour le filtrage traditionnel, nous utilisons l'algorithme d'ordre lexicographique strict de Frisch et coll. [3] qui se déclenche lorsqu'une borne

(supérieure ou inférieure) d'une variable est modifiée. La complexité de cet algorithme est $O(rb)$.

3.4.2 Filtrage tardif

Puisqu'on souhaite uniquement filtrer la variable y_{ij} qui vient d'être sélectionnée, l'algorithme peut être simplifié. Si une variable à la deuxième ligne ou plus bas est sélectionnée, alors il faut vérifier si elle est dans la première colonne ou non. Si elle est dans la première colonne, alors nous filtrons simplement son domaine dans le but de retirer les valeurs plus petites à la valeur de la variable juste au-dessus. Sinon, alors il faut imposer l'ordre lexicographique tant et aussi longtemps que les valeurs des variables à gauche de y_{ij} sont égales aux valeurs de la variable juste au-dessus (ligne $i-1$). Tout dépendant si y_{ij} est à la dernière colonne ($j = r$) ou non, la manière de filtrer son domaine change légèrement. Cette différence s'explique par le fait que lorsqu'on arrive à la dernière colonne, il ne reste plus aucune chance de pouvoir forcer l'ordre lexicographique. Voir l'algorithme 1 pour le pseudocode. La fonction Valeur retourne la valeur instanciée d'une variable et la fonction Dom retourne le domaine d'une variable. La complexité de cet algorithme est $O(r + b)$.

Algorithme 1 : LexLigneTardif

Entrée : La variable y_{ij} qui vient d'être sélectionnée
si $i > 1$ **alors**
  si $j > 1$ **alors**
    si $Valeur(y_{(i-1),l}) = Valeur(y_{il}) \quad \forall l \in [1..(j-1)]$ **alors**
      si $j \neq r$ **alors**
        $Dom(y_{ij}) = Dom(y_{ij}) \setminus \{x \mid x < Valeur(y_{(i-1),j})\}$
      sinon
        $Dom(y_{ij}) = Dom(y_{ij}) \setminus \{x \mid x \leq Valeur(y_{(i-1),j})\}$
  sinon
    `// La première colonne de la matrice doit être en ordre non décroissant`
    $Dom(y_{ij}) = Dom(y_{ij}) \setminus \{x \mid x < Valeur(y_{(i-1),j})\}$

3.5 Ordre lexicographique entre les colonnes

3.5.1 Filtrage traditionnel

Pour chaque variable qui a été instanciée dans la matrice, nous devons vérifier dans quelles lignes chaque valeur a été utilisée. Pour les valeurs qui ont été utilisées sur les mêmes lignes, nous devons seulement conserver la plus petite valeur dans le domaine de la variable suivant la variable qui vient d'être instanciée. La complexité de cet algorithme est $O(v(r + b^2))$.

3.5.2 Filtrage tardif

Pour chaque variable qui a été instanciée, nous devons vérifier dans quelles lignes chaque valeur a été utilisée. Pour les valeurs qui ont été instanciées sur les mêmes lignes, nous devons conserver uniquement la valeur la plus petite dans le domaine de la variable sélectionnée. La complexité de cet algorithme est $O(v(r + b^2))$.

4 Expérimentations

Dans cette section, nous comparons le filtrage selon qu'il réponde au filtrage classique ou tardif. Nous procédons d'abord à l'évaluation pour les BIBDs. Ensuite, à l'aide d'un contre-exemple (remplissage d'une matrice auxquelles seule la contrainte LEX est appliquée), nous montrons que le filtrage tardif n'est évidemment pas approprié pour tous les problèmes.

4.1 BIBDs

Nous utilisons des instances qui proviennent de Flener et coll. [1] ainsi que Yokoya et Yamada [18]. Les algorithmes ont été implémentés en C# étant donné que le filtrage tardif était difficile à implémenter dans un solveur générique.

Le serveur est un i5-2.60 GHz avec 8 Gb de mémoire vive.

Pour rappel, les variables sont instanciées en ordre lexicographique (gauche à droite, haut à bas). L'heuristique de choix de valeur sélectionne toujours la plus petite valeur possible en premier. Une stratégie de retour-arrière chronologique (DFS) est appliquée en cas d'échec. Le Tableau 2 montre le temps requis pour explorer complètement les arbres de recherche. Nous présentons aussi le nombre d'échecs ainsi que le nombre de solutions trouvées pour chaque instance BIBD. Comme nous pouvons le voir, le nombre d'échecs est exactement le même selon qu'on utilise une approche ou une autre. Par contre, le temps de résolution se retrouve à être réduit de 20 % à 30 % lorsque le filtrage tardif est utilisé.

Dans le but de mieux analyser les résultats présentés au Tableau 2, l'image 3 montre le temps requis pour explorer jusqu'à un certain pourcentage de l'arbre de recherche pour les deux approches. La relation entre les deux approches est linéaire parce que nous ne changeons pas la complexité des algorithmes ou le niveau de filtrage. Nous montrons ces résultats uniquement pour l'instance $(8, 4, 3)$ dans le but d'économiser de l'espace, mais les résultats sont très similaires lorsqu'on analyse les autres instances.

Nous avons également cherché à déterminer si le gain est distribué uniformément sur l'ensemble des

Tableau 2 – Résultats selon qu'on filtre traditionnellement ou tardivement

Instance	Type algo	Temps (s)	#Échecs	#sols
(6, 3, 2)	Filtrage traditionnel	0,097	486	1
	Filtrage tardif	0,076	486	1
(7, 3, 2)	Filtrage traditionnel	2,227	11 817	12
	Filtrage tardif	1,579	11 817	12
(9, 3, 1)	Filtrage traditionnel	1,077	6 439	2
	Filtrage tardif	0,676	6 439	2
(8, 4, 3)	Filtrage traditionnel	142,989	739 695	92
	Filtrage tardif	102,949	739 695	92
(6, 3, 4)	Filtrage traditionnel	27,280	79 994	21
	Filtrage tardif	21,625	79 994	21
(11, 5, 2)	Filtrage traditionnel	19,511	140 530	2
	Filtrage tardif	13,153	140 530	2
(7, 3, 3)	Filtrage traditionnel	349,753	941 904	220
	Filtrage tardif	266,021	941 904	220

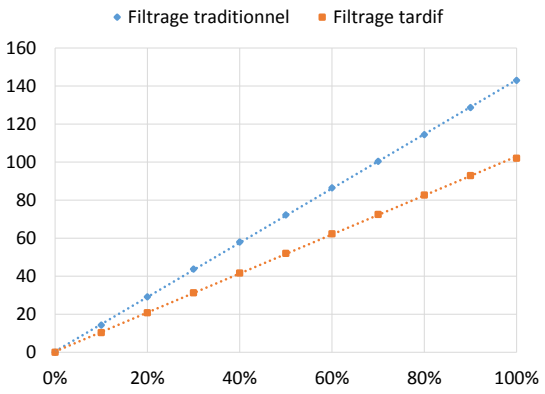

Image 3 – Temps requis en secondes à un certain % d'exploration pour l'instance $(8, 4, 3)$

contraintes. Le Tableau 3 montre comment le temps de calcul est réduit pour chaque contrainte. Nous nous serions attendus à ce qu'il puisse être préférable de filtrer au plus tôt certaines contraintes et d'autres le plus tard possible, mais ce n'est pas le cas pour le problème étudié. La différence est positive pour chaque contrainte. Par contre, l'amélioration diffère grandement d'une contrainte à une autre.

4.2 Contre-exemple : conserver seulement la contrainte d'ordre lexicographique

À titre de contre-exemple, nous avons défini un problème qui consiste à remplir une matrice et de lui appliquer la contrainte d'ordre lexicographique strict (LEX) entre les lignes. Nous effectuons le filtrage soit en version traditionnelle (Frisch et coll.) ou tardive (voir 3.4.2).

Plus précisément, nous considérons des tableaux de variables de 2 lignes et c colonnes où, selon l'instance, $c \in \{5, 6, 7\}$. Chaque variable a pour domaine l'ensemble $\{1, 2, 3\}$. Il y a une contrainte LEX entre les deux lignes.

Nous voyons que l'ordre lexicographique filtré de manière traditionnelle (Frisch et coll.) et le filtrage tardif définissent le même espace de solutions, mais que le filtrage est plus efficace avec le filtrage traditionnel (moins d'échecs et donc moins de retours-arrière).

5 Conclusion

Nous souhaitions savoir quel était le meilleur moment pour exécuter les algorithmes de filtrage lorsqu'on tente de résoudre un BIBD. Nous montrons qu'en retardant le filtrage jusqu'au dernier moment (filtrer le domaine d'une variable uniquement lorsque la variable vient d'être sélectionnée et sur le point d'être instanciée) réduit le temps de calcul entre 20 % et 30 % pour les instances présentées. L'approche a préservé le même espace solution et le même niveau de filtrage.

Il va de soi que ces résultats sont spécifiques au problème que nous avons étudié. Par contre, il est possible que d'autres genres de problèmes puissent aussi bénéficier d'un filtrage tardif. Pour certains problèmes, l'approche optimale pourrait être un mélange de filtrage classique et tardif. Les solveurs génériques actuels ne donnent pas accès à l'événement *variableAÉté-Sélectionnée*. Bien qu'il ne s'agisse pas d'une panacée, nous pensons qu'ajouter cet événement aux solveurs de PPC pourrait faciliter l'implémentation, le test et finalement l'utilisation d'algorithmes de filtrage tardif pour d'autres problèmes qui pourraient en bénéficier.

Tableau 3 – Gain en temps pour les contraintes de l'instance (8, 4, 3)

	Temps filtrage traditionnel (s)	**Temps filtrage tardif (s)**	**Réduction en %**
Contrainte ordre croissant dans ligne	6,094	2,160	64,56 %
Contrainte k	11,945	2,438	79,59 %
Contrainte λ	23,606	6,247	73,54 %
Ordre lex lignes	1,117	0,117	89,56 %
Ordre lex colonnes	89,667	75,019	16,34 %

Tableau 4 – Frisch et coll. vs filtrage tardif pour ordre lexicographique strict entre 2 lignes

#Colonnes	**Type algo**	**Temps (s)**	**#Échecs**	**#sols**
5	Frisch et coll.	0,296	0	29 403
	Filtrage tardif	0,316	81	29 403
6	Frisch et coll.	2,330	0	265 356
	Filtrage tardif	2,438	243	265 356
7	Frisch et coll.	20,648	0	2 390 391
	Filtrage tardif	21,046	729	2 390 391

Références

[1] Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming-CP 2002*, pages 462–477. Springer, 2002.

[2] Pierre Flener, Justin Pearson, and Luis G Reyna. Financial portfolio optimisation. In *International Conference on Principles and Practice of Constraint Programming*, pages 227–241. Springer, 2004.

[3] Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. In *Principles and Practice of Constraint Programming-CP 2002*, pages 93–108. Springer, 2002.

[4] Alan Frisch, Christopher Jefferson, and Ian Miguel. Symmetry breaking as a prelude to implied constraints : A constraint modelling pattern. In *ECAI*, volume 16, page 171, 2004.

[5] Gecode Team. Gecode : Generic constraint development environment, 2006. Disponible sur `http://www.gecode.org`.

[6] Takakazu Kurokawa and Yoshiyasu Takefuji. Neural network parallel computing for bibd problems. *IEEE Transactions on Circuits and Systems II : Analog and Digital Signal Processing*, 39(4) :243–247, 1992.

[7] François Pachet and Pierre Roy. Automatic generation of music programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 331–345. Springer, 1999.

[8] Steven Prestwich. Balanced incomplete block design as satisfiability. In *Proceedings of the 12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.

[9] Steven Prestwich. *A local search algorithm for balanced incomplete block designs*, pages 132–143. Springer, 2003.

[10] Patrick Prosser and Evgeny Selensky. *A study of encodings of constraint satisfaction problems with 0/1 variables*, pages 121–131. Springer, 2003.

[11] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. Disponible sur `http://www.choco-solver.org`.

[12] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[13] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1*, pages 209–215. AAAI Press, 1996.

[14] Christian Schulte and Peter J Stuckey. When do bounds and domain propagation lead to the

same search space ? *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3) :388–425, 2005.

[15] Christian Schulte and Peter J Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1) :2, 2008.

[16] Christian Schulte and Guido Tack. Implementing efficient propagation control. *TRICS*, 2010.

[17] Douglas R. Stinson. *Combinatorial Designs : Constructions and Analysis*. Springer, New York, 2004.

[18] Daisuke Yokoya and Takeo Yamada. A mathematical programming approach to the construction of bibds. *International Journal of Computer Mathematics*, 88(5) :1067–1082, 2011.