# Des encodages PB pour la résolution de problèmes CSP

Thibault Falque, Romain Wallon

# Des encodages PB pour la résolution de problèmes CSP

Thibault Falque[1,2], Romain Wallon[2]

[1] Exakis Nelite
[2] CRIL, Univ Artois & CNRS

{falque,wallon}@cril.univ-artois.fr

## Résumé

*Une approche possible pour résoudre un problème CSP est d'encoder ce problème sous la forme d'une formule CNF, et ensuite utiliser un solveur SAT pour la résoudre. Le principal avantage de cette technique est qu'elle permet de bénéficier de l'efficacité pratique des solveurs SAT modernes, fondés sur l'architecture CDCL. Cependant, le pouvoir d'inférence de ces solveurs est assez « faible », car limité par celui du système de preuve par résolution utilisé pendant l'analyse de conflit. Cette observation a conduit au développement de solveurs pseudo-booléen (PB), qui implantent le système de preuve plus puissant des plans-coupes, ainsi que de nombreuses autres techniques héritées des solveurs SAT. De plus, les solveurs PB sont capables de raisonner nativement sur des contraintes PB, c'est-à-dire, des équations ou inéquations linéaires en variables booléennes. Ces contraintes sont plus concises que les clauses, de sorte qu'une seule contrainte PB peut représenter un nombre exponentiel de clauses. Dans cet article, nous tirons parti à la fois de cette concision et du pouvoir d'inférence des solveurs PB pour résoudre des problèmes CSP. Pour ce faire, nous définissons des encodages PB pour différentes contraintes populaires, et confions leur résolution à des solveurs PB pour comparer leurs performances à celles d'autres solveurs CSP existants.*

## Mots-clés

*programmation par contraintes, encodages SAT, solveurs PB*

## Abstract

*One of the possible approaches for solving a CSP is to encode the input problem into a CNF formula, and then use a SAT solver to solve it. The main advantage of this technique is that it allows to benefit from the practical efficiency of modern SAT solvers, based on the CDCL architecture. However, the reasoning power of SAT solvers is somehow "weak", as it is limited by that of the resolution proof system they use internally. This observation led to the development of so called pseudo-Boolean (PB) solvers, that implement the stronger cutting planes proof system, along with many of the solving techniques inherited from SAT solvers. Additionally, PB solvers can natively reason on PB constraints, i.e., linear equalities or inequalities over Boolean variables. These constraints are more succinct than clauses, so that a single PB constraint can represent exponentially many clauses. In this paper, we leverage both this succinctness and the reasoning power of PB solvers to solve CSPs by designing PB encodings for different common constraints, and feeding them into PB solvers to compare their performance with that of existing CP solvers.*

## Keywords

*constraint programming, PB solving, SAT encodings*

# 1 Introduction

Le problème de satisfaction de contraintes (CSP) consiste à déterminer si un ensemble de contraintes est cohérent et, si tel est le cas, d'identifier une solution satisfaisant l'ensemble de ces contraintes. Afin de résoudre ce type de problèmes, plusieurs approches ont été proposées [18]. L'une de ces approches consiste à travailler nativement sur les contraintes du problème, à l'aide de structures de données efficaces permettant de représenter ces contraintes et la structure du problème, à l'image de solveurs tels que `Choco` [23], `Nacre` [11] ou encore `ACE` [1].

Une autre approche possible consiste à tirer parti de l'efficacité pratique des solveurs SAT modernes, fondés sur l'architecture CDCL [20, 21, 8] pour résoudre les problèmes CSP, qui sont eux-même NP-complets. Comme ces solveurs prennent en entrée une formule propositionnelle en forme normale conjonctive, il est nécessaire d'utiliser des *encodages* permettant de représenter les domaines des variables ainsi que les contraintes du problème initial sous forme de clauses. Dans ce cadre, de nombreuses solutions ont été proposées, comme celles décrites dans [26, 24, 10, 6, 3, 25, 1].

Malgré leur efficacité pratique, les solveurs SAT souffrent d'un problème majeur : le système de preuves par résolution, utilisé lors de l'analyse de conflit, présente un pouvoir d'inférence relativement faible, qui empêche de résoudre efficacement des problèmes en apparence simple, tel que celui du *pigeonhole principle* (ou *principe des tiroirs* en français) [13]. Plus généralement, les problèmes nécessitant de « savoir compter » sont souvent difficile à résoudre pour ce type de solveurs. Cette observation a conduit au développement de solveurs dit pseudo-booléens (PB) [7, 16, 9], qui héritent de nombreuses fonctionnalités des solveurs SAT

1. `https://github.com/xcsp3team/ace`

tout en implantant le système de preuve des plans-coupes plus puissant que celui de la résolution [12, 14, 22].
Ces solveurs sont par ailleurs capables de gérer nativement des contraintes PB, c'est-à-dire, des équations ou inéquations linéaires en variables booléennes. Or, parmi les encodages SAT existants, nombreux sont ceux qui utilisent en pratique une représentation intermédiaire des contraintes sous la forme de contraintes PB, avant d'encoder ces même contraintes sous la forme de clauses. Cette étape est nécessaire à l'utilisation d'un solveur SAT, mais requiert l'introduction de variables additionnelles, et l'augmentation du nombre de contraintes à donner au solveur (une unique contrainte PB pouvant représenter un nombre exponentiel de clauses).
Dans cet article, nous présentons donc des encodages tirant parti à la fois de la concision des contraintes PB et du pouvoir d'inférence des solveurs PB. Ces encodages exploitent notamment des approches classiques de représentation des domaines, tels que le *direct-encoding*, le *log-encoding* ou encore l'*order-encoding* pour ensuite encoder sous la forme de contraintes PB différentes contraintes populaires. Pour cela, nous commençons par introduire quelques préliminaires relatifs à la résolution de problèmes CSP et PB, avant de présenter formellement nos encodages et de les évaluer empriquement sur différents jeux d'instances.

# 2 Préliminaires

## 2.1 Programmation par contraintes

Un *réseau de contraintes* (ou *CN* pour *Constraint Network*) se compose d'un ensemble fini de variables et d'un ensemble fini de contraintes. Chaque variable $X$ peut prendre sa valeur dans un ensemble fini appelé *domaine* de $X$, et noté $\texttt{dom}(X)$. Chaque contrainte est défini par une relation sur un ensemble de variables. Une *solution* d'un CN est une affectation de valeurs à toutes les variables telle que toutes les contraintes soient satisfaites. Un CN est *cohérent* s'il admet au moins une solution, et le problème de décision correspondant, appelé *CSP* (pour *Constraint Satisfaction Problem*), consiste à déterminer si un CN est cohérent ou non.

## 2.2 Solveurs SAT et CSP

Une variable $x$ est dite *booléenne* lorsque $\texttt{dom}(X) = \{0, 1\}$. Nous appelons *littéral* $\ell$ une variable booléenne $x$ ou sa négation $\bar{x} = 1 - x$. Le littéral $\ell$ est dit *satisfait* lorsque $\ell$ est affecté à 1, et *falsifié* dans le cas contraire. Une *clause* est une disjonction de littéraux, qui impose la satisfaction d'au moins l'un de ses littéraux, et un problème est dit en *Forme Normale Conjonctive* (*CNF*, pour *Conjunctive Normal Form*) lorsqu'il est constitué d'une conjonction de clauses. Le *problème de cohérence propositionnel* (*SAT*) consiste à déterminer si une telle conjonction possède une solution.
Le problème SAT étant le problème NP-complet de référence [5], il est possible d'utiliser des solveurs SAT pour résoudre des problèmes CSP, en utilisant différents encodages. En particulier, pour représenter le domaine d'une variable CSP $X$, nous pouvons dans un premier temps utiliser le *direct-encoding* (voir par exemple [26]). Celui-ci consiste à utiliser une variable booléenne $x_v$ pour chacune des valeurs $v \in \texttt{dom}(X)$. Dans ce cas, la valeur affectée à la variable $X$ peut être obtenue en identifiant (l'unique) variable booléenne $x_v$ satisfaite.
Cette représentation est particulièrement commode dans le cas d'un domaine où les valeurs sont énumérées et ne constituent pas un intervalle de valeurs. Dans ce dernier cas, une autre solution consiste à représenter la variable à l'aide du *log-encoding*, qui utilise la décomposition en base 2 de la variable $X$ à l'aide de variables booléennes $b_i$ représentant les bits de $X$ (voir par exemple [26]). Cette représentation est définie par l'égalité suivante :

$$X = \min(\texttt{dom}(X)) + \sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i$$

Notons ici l'ajout de $\min(\texttt{dom}(X))$, qui vise à garantir que la décomposition en base 2 encode toujours une valeur valant au minimum 0.
Enfin, une dernière représentation possible est celle fondée sur l'*order-encoding* [25], qui utilise une variable booléenne $x_{\geq v}$ pour chacune des valeurs $v \in \texttt{dom}(X) \setminus \{\min(\texttt{dom}(X))\}$, qui est satisfaite si et seulement si $X \geq v$. Dans ce cas, la valeur affectée à $X$ peut être obtenue en identifiant deux variables $x_{\geq v}$ et $x_{\geq v+1}$ telle que la première est satisfaite et la seconde est falsifiée, auquel cas la variable $X$ est affectée à $v$.

## 2.3 Contraintes pseudo-booléennes (PB)

Une *contrainte pseudo-booléenne* (*PB*) est une contrainte de la forme $\sum_{i=1}^{n} \alpha_i \ell_i \ \triangle \ \delta$, où $n$ est un entier naturel, les *poids* (ou *coefficients*) $\alpha_i$ et le *degré* $\delta$ sont des entiers, les $\ell_i$ sont des littéraux et $\triangle \in \{<, \leq, =, \geq, >\}$.
Une contrainte PB est dite *normalisée* lorsque les coefficients et le degré de cette contrainte sont strictement positifs, et $\triangle$ est $\geq$. Toute contrainte PB peut être écrite sous la forme d'une conjonction de contraintes PB normalisées, ce qui peut être particulièrement commode dans le cas des encodages que nous présentons plus loin.
Une *contrainte de cardinalité PB* est une contrainte PB normalisée dont tous les coefficients sont égaux à 1, et une *clause* est une contrainte de cardinalité PB de degré 1. Cette définition coïncide avec la définition de clause en tant que disjonction de littéraux, et illustre le fait que les solveurs PB généralisent les solveurs SAT.

# 3 Encodages purement PB

Dans cette section, nous exploitons le *direct-encoding*, le *log-encoding* et l'*order-encoding* pour représenter le domaine des variables d'un CN, et utilisons cette représentation pour encoder sous forme de contraintes PB plusieurs contraintes CSP couramment utilisées, en considérant tout particulièrement celles reconnues par les *mini-solvers* de la compétition XCSP3, ainsi des contraintes de comptage.

## 3.1 (Dés)activation de contraintes PB

Pour encoder un problème CSP (qu'il s'agisse du domaine de ses variables ou de ses contraintes) sous la forme de contraintes PB, il est souvent nécessaire de pouvoir activer (ou désactiver) une contrainte. Pour ce faire, il est commode d'utiliser un *sélecteur*, c'est-à-dire, une variable fraîche $s$ dont la satisfaction entraîne celle de la contrainte considérée. Dans le cas des contraintes PB, un tel sélecteur $s$ pourrait avoir la sémantique suivante (où $\Rightarrow$ représente l'implication matérielle) :

$$s \Rightarrow \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$$

La forme particulière des contraintes PB fournit une manière simple et concise de représenter cette implication sous la forme d'une unique contrainte PB, donnée ci-dessous :

$$\delta \bar{s} + \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$$

Rappelons que, dans le cas présenté ci-dessus, la satisfaction de la contrainte ne garantit pas la satisfaction de la variable $s$. Lorsque cette garantie est nécessaire, il faut ajouter l'implication réciproque :

$$s \Leftarrow \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$$

Cette implication peut elle aussi être représentée sous la forme d'une unique contrainte pseudo-booléenne, à savoir :

$$\left(\sum_{i=1}^{n} \alpha_i - \delta + 1\right) s + \sum_{i=1}^{n} \alpha_i \bar{\ell_i} \geq \sum_{i=1}^{n} \alpha_i - \delta + 1$$

Dans la suite, nous noterons $s$ un sélecteur pour lequel uniquement la première implication est définie, et $\mathcal{S}$ un sélecteur pour lequel les deux implications sont définies (si nécessaire, ces symboles pourront être indicés).
Pour illustrer l'utilisation des sélecteurs, considérons la contrainte de diséquation suivante :

$$\sum_{i=1}^{n} \alpha_i \ell_i \neq \delta$$

Notons que cette contrainte ne peut pas être normalisée directement, comme l'opérateur $\neq$ n'est pas autorisé par les contraintes PB. En revanche, nous pouvons observer que cette contrainte est en fait équivalente à la disjonction :

$$\left(\sum_{i=1}^{n} \alpha_i \ell_i \leq \delta - 1\right) \vee \left(\sum_{i=1}^{n} \alpha_i \ell_i \geq \delta + 1\right)$$

Nous définissons alors deux nouveaux sélecteurs, $s_{\leq}$ et $s_{\geq}$, ayant respectivement les sémantiques suivantes :

$$s_{\leq} \Rightarrow \sum_{i=1}^{n} \alpha_i \ell_i \leq \delta - 1$$

$$s_{\geq} \Rightarrow \sum_{i=1}^{n} \alpha_i \ell_i \geq \delta + 1$$

Il suffit alors d'ajouter la définition de ces deux sélecteurs, ainsi que la disjonction $s_{\leq} \vee s_{\geq}$, pour représenter la diséquation présentée plus haut.

## 3.2 Les variables et leurs domaines

Afin de nous assurer que les encodages introduit dans la section précédente représente effectivement le domaine des variables du problème original, nous avons besoin d'ajouter un certain nombre de contraintes. Dans le cas du *direct-encoding*, il suffit d'ajouter la contrainte PB suivante, qui garantit que la variable $X$ peut prendre exactement une valeur parmi celles de son domaine :

$$\sum_{v \in \mathtt{dom}(X)} x_v = 1$$

Il est alors possible de représenter la variable $X$ à l'aide de l'égalité suivante :

$$X = \sum_{v \in \mathtt{dom}(X)} v x_v$$

Dans le cas du *log-encoding*, la représentation de la variable $X$ étant donnée par l'égalité suivante :

$$X = \min(\mathtt{dom}(X)) + \sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i$$

la contrainte à ajouter pour s'assurer que le domaine de $X$ soit bien respecté est donnée ci-dessous (rappelons que cet encodage est utilisé ici uniquement pour représenter des domaines correspondant à des intervalles) :

$$\sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i \leq \max(\mathtt{dom}(X)) - \min(\mathtt{dom}(X))$$

Enfin, dans le cas de l'*order-encoding*, les contraintes à ajouter sont exactement les mêmes clauses que dans l'article original introduisant cette approche [25]. Ainsi, pour toute valeur $v \in \mathtt{dom}(X) \setminus \{\min(\mathtt{dom}(X))\}$, l'implication matérielle suivante est ajoutée au solveur :

$$x_{\geq v} \Rightarrow x_{\geq v-1}$$

Lorsque le domaine de $X$ n'est pas un intervalle, il est possible d'interdire une valeur $v$ en ajoutant, en plus, l'implication :

$$x_{\geq v} \Rightarrow x_{\geq v+1}$$

Notons de plus qu'il est possible, en utilisant l'*order-encoding*, de représenter $X$ grâce à l'égalité suivante :

$$X = \min(\mathtt{dom}(X)) + \sum_{v \in \mathtt{dom}(X) \setminus \{\min(\mathtt{dom}(X))\}} x_{\geq v}$$

Comme dans le cas du *log-encoding*, remarquons ici l'ajout de $\min(\texttt{dom}(X))$ pour se ramener à l'encodage d'un domaine dont le minimum est 0. Grâce à cette représentation, il est donc possible, sans perte de généralité, de représenter toute variable CSP $X$ sous la forme d'une somme pondérée de variables booléennes, et plus généralement de littéraux, auxquels peut s'ajouter une constante $\mu$ :

$$X = \mu + \sum_{i=1}^{n} \alpha_i \ell_i$$

Notons par ailleurs que, pour encoder des contraintes CSP, il est souvent nécessaire de pouvoir déterminer si une variable est affectée à une valeur $v$ donnée. Dans le cas du *direct-encoding*, il suffit de regarder la valeur affectée à $x_v$. Pour obtenir une telle variable avec l'*order-encoding*, observons dans un premier temps que la variable $X$ est affectée à la valeur $v$ si, et seulement si, la conjonction $x_{\geq v} \wedge \overline{x_{\geq v+1}}$ est satisfaite. Pour obtenir une variable $x_v$ équivalente à celle utilisée dans le *direct-encoding*, il est possible de définir cette variable à l'aide de l'équivalence suivante :

$$x_v \Leftrightarrow x_{\geq v} \wedge \overline{x_{\geq v+1}}$$

qui peut être encodée sous la forme de contraintes PB, comme illustré dans la section précédente.
La situation est un peu plus complexe pour le *log-encoding*, car il est nécessaire de regarder l'affectation de *toutes* les variables booléennes utilisées dans la représentation de la variable pour connaître la valeur de la variable $X$. Nous proposons dans ce cas d'utiliser une forme paresseuse de *direct-encoding*, où la variable $X$ est initialement encodée *via* le *log-encoding*, puis sous la forme du *direct-encoding* uniquement si la variable $x_v$ a besoin d'être utilisée. Dans ce cas, la contrainte suivante assure la correspondance de la valeur affectée à $X$ par dans les deux encodages :

$$\sum_{v \in \texttt{dom}(X)} v x_v = \min(\texttt{dom}(X)) + \sum_{i=0}^{\lceil \log_2(X) \rceil} 2^i b_i$$

Grâce à cette approche, il est toujours possible d'obtenir, pour toute variable CSP $X$ et toute valeur $v \in \texttt{dom}(X)$ une unique variable $x_v$ représentant l'affectation $X = v$.

## 3.3 Contrainte `cardinality`

Comme nous l'avons déjà mentionné, l'un des principaux avantages des solveurs PB comparés aux solveurs SAT et leur capacité à « savoir compter » de manière relativement efficace. Pour bénéficier de cet avantage, nous proposons tout d'abord un encodage pour les contraintes `cardinality`.
Une première forme de cette contrainte consiste à imposer des bornes sur le nombre de variables parmi un ensemble $\{X_1, \ldots, X_N\}$ pouvant être affectées à une valeur $v$ donnée. Notons $m$ et $M$ le nombre minimum et le nombre maximum de variables $X_i$ pouvant être affectées à $v$, respectivement, et notons $x_v^i$ la variable booléenne représentant l'affectation $X_i = v$. Clairement, le nombre de $x_v^i$ pouvant être satisfaits doit être compris entre $m$ et $M$. Ainsi, la contrainte `cardinality` peut être représentée par la contrainte

$$m \leq \sum_{i=1}^{N} x_v^i \leq M$$

qui peut facilement se décomposer en deux contraintes PB. Notons que ces contraintes peuvent à elles seules permettre de représenter un nombre exponentiel de clauses suivant les valeurs de $m$ et $M$ [2], sans ajouter de nouvelles variables, contrairement à ce que nécessiterait un encodage CNF de ces contraintes, par exemple.
Une seconde forme de contrainte `cardinality` vise à s'assurer qu'une variable $C$ est affectée au nombre de variables $X_i$ qui sont affectées à une valeur $v$ donnée. Soit $\mu + \sum_{i=1}^{n} \alpha_i \ell_i$ la représentation PB de la variable $C$. Cette variable doit être égale au nombre de $x_v^i$ satisfait. Alors, en reprenant les mêmes notations que dans le cas précédent, cette seconde forme de contrainte de cardinalité peut être représentée à l'aide de l'égalité ci-dessous :

$$\sum_{i=1}^{N} x_v^i = \mu + \sum_{i=1}^{n} \alpha_i \ell_i$$

qui peut s'écrire sous la forme de la contrainte PB :

$$\sum_{i=1}^{N} x_v^i - \mu - \sum_{i=1}^{n} \alpha_i \ell_i = 0$$

Il existe par ailleurs des variantes des contraintes décrites dans cette section, où les valeurs ne sont pas des constantes mais des variables $Z$. Pour encoder de telles contraintes, il suffit de remplacer les variables booléennes $x_v^i$ dans les contraintes PB définies ici par des variables $x_Z^i$, telles que :

$$x_Z^i \Leftrightarrow (X_i - Z = 0)$$

Notons que, en utilisant les décompositions de $X_i$ et $Z$ sous la forme de sommes pondérées de variables booléennes, il est possible de représenter cette contrainte à l'aide des sélecteurs présentés dans la Section 3.1.

Ajoutons qu'à l'image de la contrainte `cardinality`, des encodages PB peuvent être proposés pour d'autres contraintes de comptage, telles que `count` ou `nValues`, mais ils sont omis ici. En effet, aucune des instances XCSP3 que nous avons pu trouver ne comportant de telles contraintes, nous n'avons pas été en mesure d'évaluer ces encodages, et nous ne pouvons donc fournir aucune conclusion à leur sujet à ce jour.

## 3.4 Contraintes *mini-solvers*

Nous proposons maintenant des encodages pour une partie des contraintes autorisées pour les *mini-solvers* ayant participé à la compétition XCSP'19.

### 3.4.1 Contrainte `sum`

Une contrainte `sum` est une contrainte de la forme suivante :

$$\sum_{i=1}^{N} A_i X_i \odot k$$

où $\odot \in \{<, \leq, =, \neq, \geq, <\}$. Il est clair qu'une telle contrainte peut facilement s'écrire sous la forme d'une contrainte PB. En effet, si chacun des $X_i$ s'écrit sous la forme $X_i = \mu^i + \sum_{j=1}^{n_i} \alpha_j^i x_j^i$, où les $x_j^i$ sont des variables booléennes, alors la contrainte ci-dessous est équivalente à :

$$\sum_{i=1}^{N} A_i \left( \mu^i + \sum_{j=1}^{n_i} \alpha_j^i x_j^i \right) \odot k$$

qui, après développement des facteurs constants, peut être écrite sous la forme d'une contrainte PB (le cas de l'opérateur $\neq$ est traité de manière analogue à celle présentée dans la Section 3.1).

### 3.4.2 Contrainte `allDifferent`

La contrainte globale `allDifferent` vise à garantir que, parmi un ensemble de variables, les valeurs prises par ces variables sont toutes différentes. Nous illustrons l'encodage de cette contrainte sur l'exemple suivant :

$$\texttt{allDifferent}(X_1, \ldots, X_n)$$

Soit $\mathcal{D} = \bigcup_{i=1}^{N} \texttt{dom}(X_i)$. La sémantique de la contrainte `allDifferent` impose que chaque valeur $v \in \mathcal{D}$ soit utilisée au plus une fois parmi les variables $X_i$. Soit donc $v \in \mathcal{D}$, et notons $x_v^i$ la variable booléenne représentant l'égalité $X_i = v$ pour tout $i \in 1..N$. Nous pouvons représenter cette contrainte sur $v$ par la contrainte suivante, qui est déjà une contrainte PB :

$$\sum_{\substack{i=1 \\ v \in \texttt{dom}(X_i)}}^{N} x_v^i \leq 1$$

Cette contrainte devant s'appliquer à toutes les valeurs $v \in \mathcal{D}$, il faut ensuite répéter l'opération pour chacune des valeurs possibles. Observons que cet encodage n'est pas sans rappeler celui du problème du *pigeonhole-principle*, qui est connu pour être difficile pour les solveurs SAT utilisant la résolution [13]. Ici, l'encodage sous forme de contraintes PB permet donc, grâce à l'utilisation des solveurs PB, d'améliorer l'efficacité du raisonnement sur le sous-ensemble de contraintes correspondant à la contrainte `allDifferent`.

### 3.4.3 Contrainte `extension` pour les supports

Un *support* permet de lister explicitement les solutions possibles d'une contrainte. Pour encoder de telles contraintes, commençons par noter qu'une contrainte PB de la forme $\sum_{i=1}^{n} \ell_i \geq n$ permet de représenter la conjonction des littéraux $\ell_i$, i.e., $\bigwedge_{i=1}^{n} \ell_i$.

Partant de cette observation, nous pouvons encoder un support (unique) ayant la forme suivante :

$$(X_i \mid 1 \leq i \leq N) = (v_i \mid 1 \leq i \leq N)$$

à l'aide de la contrainte PB :

$$\sum_{i=1}^{N} x_{v_i}^i \geq N$$

où $x_{v_i}^i$ est la variable booléenne représentant l'égalité $X_i = v_i$ pour tout $i \in 1..N$. Dans le cas (plus fréquent) où plusieurs tuples $t$ sont autorisés par une contrainte de type support, il nous faut ajouter un sélecteur $s_t$ pour les contraintes associées à chacun de ces tuples, pour obtenir des contraintes de la forme :

$$s_t \Rightarrow \sum_{i=1}^{N} x_{v_i}^i \geq N$$

En effet, l'ensemble des tuples autorisés doit être vu comme une disjonction. En particulier l'un des tuples doit nécessairement être affecté : il faut alors ajouter la clause $\bigvee s_t$ pour terminer l'encodage du support.

Remarquons enfin que, si le symbole $\star$ est utilisé dans les tuples autorisés à la place d'une valeur $v_i$ (pour symboliser la possibilité d'utiliser une valeur quelconque pour la variable $X_i$), il suffit d'omettre le littéral correspondant à la variable $X_i$ dans la contrainte ci-dessus.

# 4 Résultats expérimentaux

Cette section présente quelques résultats expérimentaux relatifs à l'utilisation des encodages pseudo-booléens présenté dans cet article sur différents ensembles de problèmes de satisfaction. Afin d'évaluer les performances de notre approche, nous avons implanté nos encodages dans le solveur `Sat4j` [16], et exécuté plusieurs variantes de ce solveur, notées `Sat4j` + $\mathcal{S}$ dans la suite de cette section, où $\mathcal{S}$ représente le nom la variante utilisée. Lorsque aucun encodage n'est spécifié, c'est la combinaison du *direct-encoding* et du *log-encoding* qui est utilisé pour représenter les domaines des variables. Le solveur `Sat4j` + `OrderEncodingBothPOS2020` utilise quant à lui l'*order-encoding*, tandis que `Sat4j` + `OrderEncodingPrimitiveBothPOS2020` exploite en plus cet encodage pour encoder les contraintes primitives de manière plus efficaces (à la manière de ce qui est proposé dans [25]). Dans le même temps, nous avons exécuté le solveur PB `RoundingSat` [9] sur l'encodage PB fournit par notre implantation dans `Sat4j`.

Nous comparons ces implantations avec différents solveurs CSP de l'état de l'art, à savoir `ACE`[2] – le nouvel avatar d'`AbsCon` – `Choco` [23] et `sCOP` [25]. Nous avons également exécuté le solveur CSP proposé avec la bibliothèque `Sat4j`, et noté `Sat4j` + `SAT` [6].

2. https://github.com/xcsp3team/ace

Tous les solveurs ont été lancés sur un cluster de machines équipées 32 Go de RAM et de 2 processeurs quadricœur Intel Xeon X5550 cadencé à 2.66 GHz. Le temps d'exécution était limité à 1200 secondes.

Dans le cadre de nos expérimentations, nous avons utilisé deux benchmarks, composés de problèmes de satisfaction provenant de la distribution XCSP [4, 19]. Le premier, noté $\mathcal{I}_{\texttt{card}}$, se compose de 5 familles de problèmes et de 145 instances. Il est composé d'instances CSP comportant des contraintes de type `cardinality` provenant du site XCSP[3]. Pour obtenir cet ensemble, nous avons dû retirer certaines instances qui comportaient des contraintes qui n'étaient pas gérées par notre approche (en particulier, des contraintes `lex`). Le second, noté $\mathcal{I}_{\texttt{XCSP18-19}}$, correspond à l'ensemble des instances des compétitions XCSP18 et XCSP19 utilisées dans le *track mini-solvers*, résultant en 32 familles de problèmes et 371 instances.

Nous présentons ci-dessous l'étude expérimentale de notre approche sur ces deux benchmarks. Ces analyses ont été réalisées avec Metrics[4], un outil d'analyse expérimentale garantissant la reproductibilité des résultats.

### 4.1 Etude de $\mathcal{I}_{\texttt{card}}$

La Figure 1 présente un aperçu des performances des différents solveurs exécutés sur l'ensemble d'instance $\mathcal{I}_{\texttt{card}}$.

Cette figure est un *cactus-plot*. Chacune des lignes correspond à un solveur, et permet de connaître le nombre d'instances résolues en un temps donné par ce solveur. Ici, nous pouvons observer que le solveur `RoundingSat` parvient à résoudre plus d'instances que les autres solveurs, en résolvant par exemple 16 instances de plus que le solveur `ACE` dans la limite des 1200 secondes. Nous pouvons de plus remarquer que les solveurs utilisant un encodage SAT, en l'occurence, `Sat4j` + `SAT` et `sCOP`, qui utilisent respectivement le *direct-encoding* et l'*order-encoding* ont des difficultés sur ces instances qui requiert des capacités de comptage (rappelons que ces instances contiennent des contraintes `cardinality`).

Afin de comparer plus précisément les deux premiers solveurs du *cactus-plot*, nous traçons à la Figure 2 un *scatter-plot* comparant `ACE` et `RoundingSat`.

Ici, chaque point représente une instance, et sa couleur la famille de problèmes à laquelle elle appartient. L'abscisse d'un point correspond au temps d'exécution de `RoundingSat` sur cette instance, et son ordonnée celui de `ACE` sur cette même instance. Cette figure montre clairement que `RoundingSat` parvient à résoudre très efficacement les instances de la famille `CarSequencing` (qui comportent essentiellement des contraintes `cardinality` et `sum`). De manière générale, nous avons pu observer que ce comportement se généralisait aux différents solveurs PB considérés dans notre étude, même si les différentes variantes de `Sat4j` restent moins efficaces que `RoundingSat`.

3. https://xcsp.org
4. https://github.com/crillab/metrics

### 4.2 Etude de $\mathcal{I}_{\texttt{XCSP18-19}}$

Afin d'évaluer les capacités de notre approche à résoudre des problèmes plus généraux, nous l'évaluons maintenant sur les instances de l'ensemble $\mathcal{I}_{\texttt{XCSP18-19}}$, qui contient une plus large variété de problèmes et de contraintes. Pour les contraintes des *mini-solvers* que nous n'avons pas présentées dans la Section 3, nous utilisons soit une combinaison classique des contraintes présentées dans cet article, soit un encodage sous forme de clauses (qui reprend donc un encodage existant, dépendant de l'encodage choisi pour les domaines des variables concernée). De cette manière, notre implantation est capable d'encoder toutes les instances de l'ensemble $\mathcal{I}_{\texttt{XCSP18-19}}$, La Figure 3 nous présente une vue globale des solveurs exécutés sur cet ensemble.

Le premier solveur, avec une belle avance sur les autres, est `ACE`. Il est suivi par `Choco` et `sCOP` respectivement en seconde et troisième place. Le premier solveur `Sat4j` arrive en quatrième position. Il s'agit de `Sat4j`+`Resolution` : ce solveur PB réalise en fait une analyse de conflit à la manière d'un solveur SAT classique, en inférant une clause de manière paresseuse chaque fois qu'une contrainte PB est rencontrée pendant l'analyse. L'avantage de cette approche est qu'elle permet de combiner la concision des contraintes PB et l'efficacité des structures de données des solveurs SAT.

Néanmoins, l'efficacité de ce solveur ne permet pas de généraliser les bonnes performances observées sur l'ensemble $\mathcal{I}_{\texttt{card}}$. Cela peut en partie s'expliquer par le fait que de (trop) nombreuses contraintes doivent encore être encodées à l'aide de clauses pour les problèmes considérés, ce qui ne permet pas d'exploiter la pleine puissance du système de preuves implanté par les solveurs PB.

Nous pouvons de plus observer que l'ordre des solveurs PB n'est pas le même que pour $\mathcal{I}_{\texttt{card}}$. Cela peut s'expliquer par la complémentarité des approches implantées par les différents solveurs comme cela a notamment été décrit dans [15, 17].

## 5 Conclusion

Dans cet article nous avons proposé d'exploiter différents encodages booléens des domaines des variables d'un problème de satisfaction de contraintes pour définir des encodages sous forme de contraintes pseudo-booléennes. Nous avons tout particulièrement considéré les contraintes reconnues par les *mini-solvers* de la compétition XCSP3, ainsi que différentes variantes de la contrainte `cardinality`. Le principal avantage des encodages proposés est qu'ils permettent d'exploiter le pouvoir d'inférence des solveurs PB, et notamment leur capacité à compter efficacement. L'analyse expérimentale a montré que nos encodages, combinés à l'utilisation de solveurs PB, permet en effet de résoudre efficacement des problèmes composés principalement de contraintes `sum` et de contraintes `cardinality`. Néanmoins, ces bonnes performances ne se généralisent pas à des problèmes comportant d'autres types de contraintes, pour lesquels les solveurs CP natifs et les solveurs fondés sur SAT restent plus performant.

A ce jour, notre approche ne permet de résoudre que

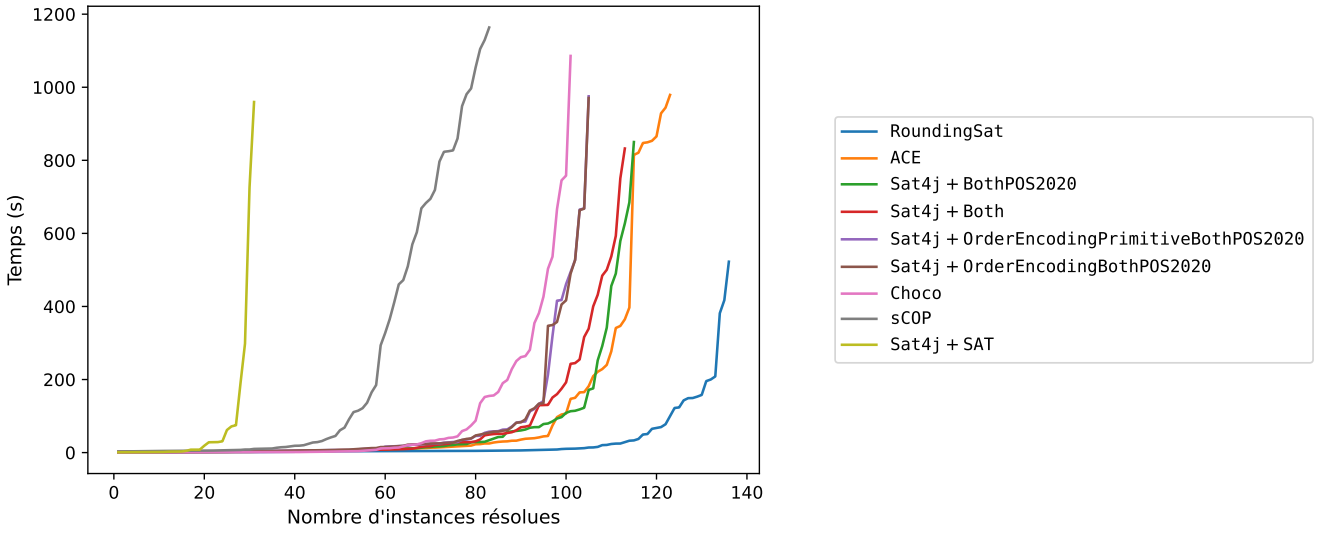


FIGURE 1 – *Cactus-plot* des solveurs sur l'ensemble d'instances $\mathcal{I}_{\texttt{card}}$.

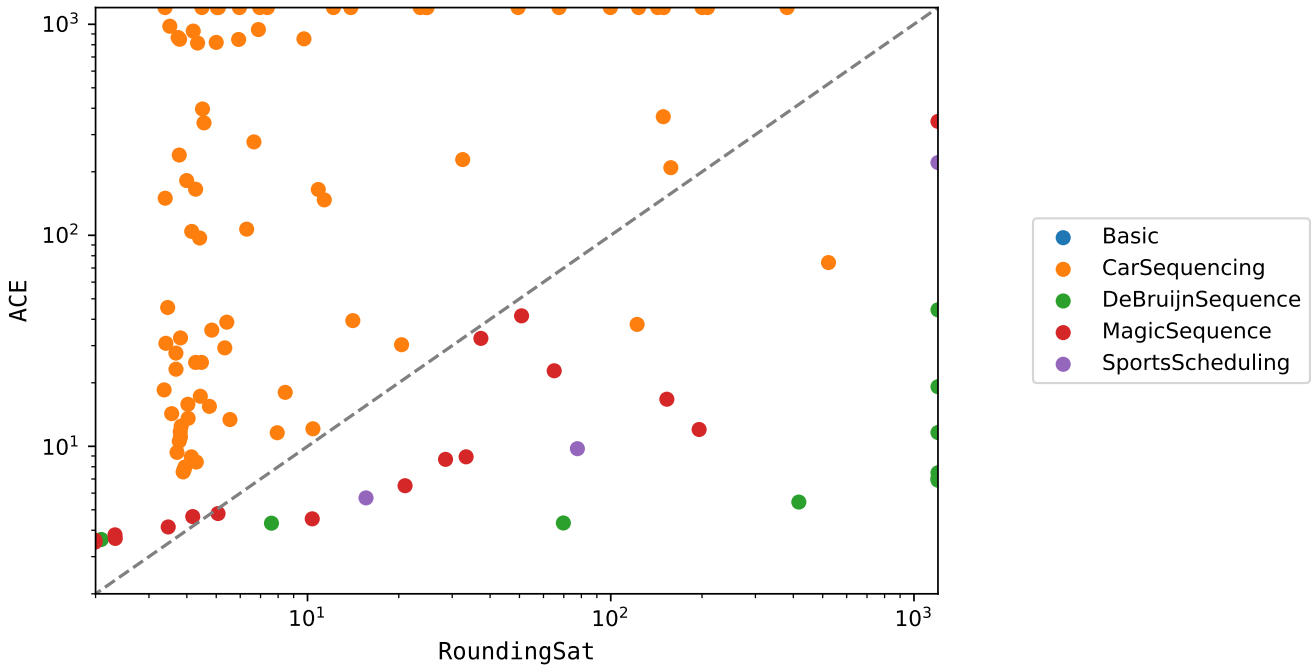


FIGURE 2 – *Scatter-plot* comparant les temps d'exécution (en secondes) d'`ACE` et de `RoundingSat` sur l'ensemble $\mathcal{I}_{\texttt{card}}$.

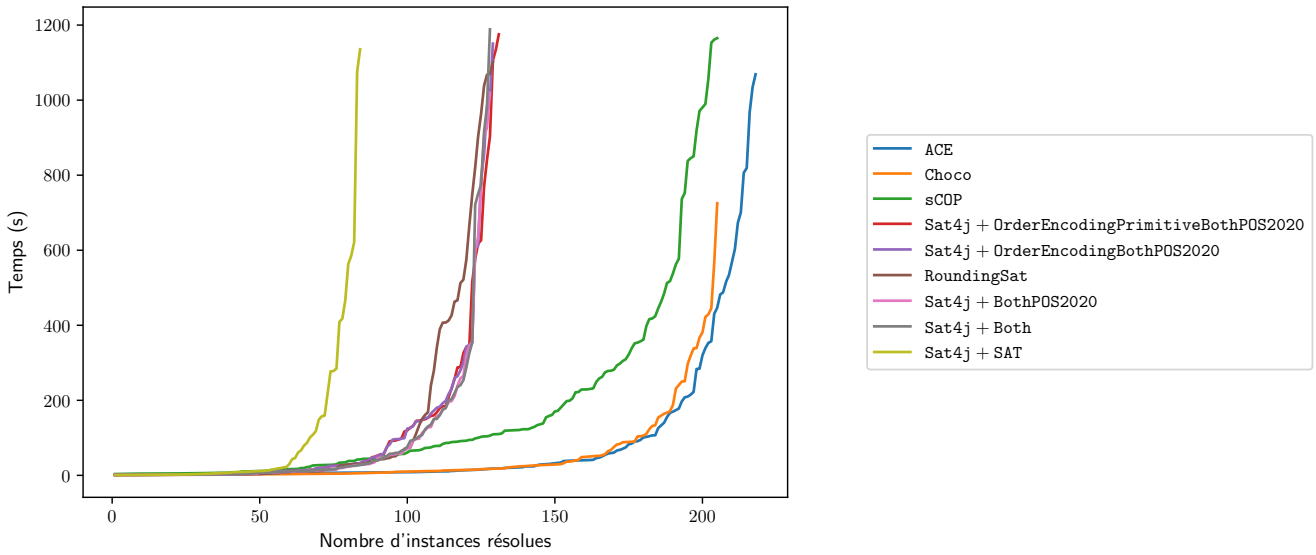


FIGURE 3 – *Cactus-plot* des solveurs sur l'ensemble d'instances $\mathcal{I}_{\texttt{XCSP18-19}}$

des problèmes comportant un sous-ensemble restreint de contraintes. A court terme, nous envisageons de définir des encodages pour d'autres types de contraintes, afin de pouvoir soumettre notre approche à la prochaine compétition XCSP. Nous souhaiterions de plus étudier de nouveaux encodages pour améliorer les performances des solveurs PB dans la résolution de problèmes CSP. A plus long terme, nous envisageons d'exploiter la complémentarité des différents paradigmes de résolution de CSP (natif, fondé sur SAT ou fondé sur PB) pour tirer le meilleur de chacune de ces approches.

## Références


[1] Ramón Béjar, Cèsar Fernández, and Francesc Guitart. Encoding basic arithmetic operations for sat-solvers. In René Alquézar, Antonio Moreno, and Josep Aguilar-Martin, editors, *Artificial Intelligence Research and Development - Proceedings of the 13th International Conference of the Catalan Association for Artificial Intelligence, l'Espluga de Francolí, Tarragona, Spain, 20-22 October 2010*, volume 210 of *Frontiers in Artificial Intelligence and Applications*, pages 239–248. IOS Press, 2010.

[2] Belaid Benhamou, Lakhdar Sais, and Pierre Siegel. Two proof procedures for a cardinality based language in propositional calculus. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*, volume 775 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1994.

[3] Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Decompositions of all different, global cardinality and related constraints. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 419–424, 2009.

[4] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3-core : A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020.

[5] Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[6] Ines Lynce Daniel Le Berre. Csp2sat4j : A simple csp to sat translator. In *Proceedings of the second CSP Competition*, 2008.

[7] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *AAAI'02*, pages 635–640, 2002.

[8] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[9] Jan Elffers and Jakob Nordström. Divide and conquer : Towards faster pseudo-boolean solving. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 1291–1299, 2018.

[10] Marco Gavanelli. The log-support encoding of CSP into SAT. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2007.

[11] Gaël Glorian. Nacre. In *Solver Descriptions of XCSP3 Competition 2018*, 2018.

[12] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, pages 275–278, 1958.

[13] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39 :297 – 308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[14] J. N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12(1) :217–239, 1988.

[15] Daniel Le Berre, Pierre Marquis, and Romain Wallon. On weakening strategies for PB solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 322–331. Springer, 2020.

[16] Daniel Le Berre and Anne Parrain. The SAT4J library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010.

[17] Daniel Le Berre and Romain Wallon. On dedicated cdcl strategies for pb solvers. In *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Proceedings*, page to appear, 2021.

[18] C. Lecoutre. *Constraint Networks : Techniques and Algorithms*. ISTE/Wiley, 2009.

[19] Christophe Lecoutre and Nicolas Szczepanski. PYCSP3 : modeling combinatorial constrained problems in python. *CoRR*, abs/2009.00326, 2020.

[20] Joao Marques-Silva and Karem A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, pages 220–227, 1999.

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[22] Jakob Nordström. On the Interplay Between Proof Complexity and SAT Solving. *ACM SIGLOG News*, 2(3) :19–44, August 2015.

[23] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco solver documentation. *TASC, INRIA Rennes, LINA CNRS UMR*, 6241, 2016.

[24] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 590–603. Springer, 2006.

[25] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2) :254–272, 2009.

[26] Toby Walsh. SAT v CSP. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.