Réutiliser ou adapter les prouveurs SAT pour l'optimisation booléenne

Daniel Le Berre **Emmanuel Lonca** *

Université Lille Nord de France, F-59000 Lille, France
Université d'Artois, CRIL, F-62300 Lens, France
CNRS, UMR8188, F-62300 Lens, France
{leberre,lonca}@cril.fr

Abstract

D'années en années, le succès des approches basées sur SAT dans de nombreux domaines a contribué à la création de solveurs SAT robustes, efficaces, et pensés pour être réutilisables dans des applications tierces. Dans de nombreux cas, ces applications utilisent l'interface incrémentale des prouveurs SAT proposée par Minisat, qui leur permet d'utiliser les solveurs SAT comme des boites noires, c'est à dire sans se préoccuper de leur fonctionnement interne. Si cette approche semble limitée à la résolution de problèmes de décision, elle s'avère aussi utilisée avec succès dans le cadre de problèmes d'optimisation en variables booléennes, tels que l'optimisation pseudo-booléenne ou MAXSAT. Il parait cependant plus intéressant dans ce contexte de permettre au solveur de poursuivre sa recherche quand il trouve une solution, plutôt que de le laisser s'arrêter et commencer la résolution d'un nouveau problème de décision. Cela nécessite au niveau du prouveur SAT de pouvoir recevoir de nouvelles contraintes durant la recherche. C'est ce qui se passe par exemple dans le cadre des prouveurs SMT, ou lors de la génération paresseuse de contraintes en CSP : un prouveur SAT est adapté pour recevoir des clauses pendant la recherche. Dans le cadre des problèmes d'optimisation qui nous intéressent, on souhaite pouvoir créer de manière paresseuse des contraintes de borne, c'est à dire ajouter des contraintes de cardinalité ou des contraintes pseudo-booléennes pendant la recherche. Nous avons intégré cette fonctionnalité sur la plate-forme d'optimisation booléenne libre Sat4j, et intégré un algorithme d'optimisation par contraintes de bornes paresseuses. Nous comparons l'approche boite noire et l'approche paresseuse sur l'ensemble des problèmes d'optimisation issus des compétitions PB10 et MAXSAT10. On n'observe pas de différences flagrantes de performance (en terme de nombre d'instances résolues et de temps d'exécution) entre les deux approches, malgré des comportements différents. Nous montrons que l'approche par ajout de contraintes à la volée peut cependant s'avérer intéressante, par exemple dans le cadre de l'énumération de solutions.

*Ce travail est financé en partie par le conseil régional Nord-Pas de Calais et le programme FEDER.

1 Introduction

D'années en années, le succès des approches basées sur SAT dans de nombreux domaines comme la vérification de matériel [6], la vérification de logiciels [13], la planification [21] ont contribué à la création de solveurs SAT robustes, efficaces, et pensés pour être réutilisables dans des applications tierces. Dans de nombreux cas, ces applications utilisent l'interface incrémentale des prouveurs SAT proposée par Minisat [8, 9], qui leur permet d'utiliser les solveurs SAT comme des boites noires, c'est à dire sans se préoccuper de leur fonctionnement interne (bien que l'état interne de ces solveurs puisse évoluer au fur et à mesure de leur utilisation, notamment en ce qui concerne la base des clauses apprises, mais sans le contrôle de l'utilisateur). Si cette approche semble limitée à la résolution de problèmes de décision, elle s'avère aussi utilisée avec succès dans le cadre de problèmes d'optimisation en variables booléennes, tels que l'optimisation pseudo-booléenne [22] ou MAXSAT[15]. Il parait cependant plus intéressant dans ce contexte de permettre au solveur de poursuivre sa recherche quand il trouve une solution, plutôt que de le laisser s'arrêter et commencer la résolution d'un nouveau problème de décision. Cela nécessite l'introduction au niveau du prouveur SAT d'un mécanisme permettant de pouvoir recevoir de nouvelles contraintes durant la recherche. C'est

ce qui se passe par exemple dans le cadre des prouveurs SMT [4], ou lors de la génération paresseuse de contraintes en CSP [20] : un prouveur SAT est adapté de manière à pouvoir recevoir des clauses pendant la recherche. Dans le cadre des problèmes d'optimisation qui nous intéressent, on souhaite pouvoir créer de manière paresseuse des contraintes de borne, c'est-à-dire ajouter des contraintes de cardinalité ou des contraintes pseudo-booléennes pendant la recherche. L'idée n'est elle-même pas nouvelle, puisque dès l'apparition des prouveurs SAT dirigés par les conflits, des solveurs basés sur ce principe ont vu le jour (comme par exemple BSOLO [16]). Cependant, les prouveurs « spécialisés » conçus il y a une dizaine d'année ont tendance à disparaître au profit de ceux qui réutilisent sans modifications les solveurs SAT (c'est particulièrement évident ces dernières années pour la résolution du problème MAXSAT [10, 18, 2]). Ce qui nous intéresse ici est d'évaluer l'impact (positif ou négatif) d'une communication privilégiée avec le solveur SAT par rapport à une approche boite noire sur une plateforme commune, et de définir précisément quelles sont les conditions nécessaires pour permettre cette communication sans remettre en cause le fonctionnement du solveur.

Nous avons intégré l'ajout de contraintes à la volée sur la plate-forme d'optimisation booléenne libre Sat4j, et intégré un algorithme d'optimisation par contraintes de bornes paresseuses. Nous comparons l'approche boite noire utilisée jusqu'à présent dans Sat4j et l'approche paresseuse sur l'ensemble des problèmes d'optimisation issus des compétitions PB10 et MAXSAT10. On n'observe pas de différence flagrante de performance (en terme de nombre d'instances résolues et de temps d'exécution) entre les deux approches, ce qui s'explique à la fois par les particularités des benchmarks de ces compétitions et par la spécificité du processus d'optimisation. On observe cependant des comportements particuliers, notamment en terme de nombre de solutions intermédiaires rencontrées lors du calcul de la valeur optimale de la fonction objectif sur certaines instances.

Nous présentons tout d'abord l'observation qui a motivé cette étude : l'amélioration significative des performances de l'énumération de solutions basée sur la production de clauses paresseuses. Nous présentons ensuite le cadre de notre travail : l'optimisation en variables booléennes et les approches par résolution successive de problèmes de satisfaction de contraintes booléennes. Nous détaillons ensuite les conditions nécessaires pour l'apprentissage de contraintes dynamiques. Nous présentons ensuite les résultats expérimentaux et concluons par quelques perspectives.

2 Motivation

Nous utilisons les prouveurs Abscon[17] et Sat4j[5] pour illustrer les forces et les faiblesses des approches SAT et CSP pour la résolution de divers problèmes académiques à nos étudiants de Master. Sat4j-CSP traduit les problèmes CSP en SAT par l'encodage support pour les contraintes binaires [11] et direct [27] pour les contraintes n-aires . Afin de fournir des fonctionnalités similaires à Abscon, il était nécessaire d'ajouter l'énumération de solutions à Sat4j-CSP. Sat4j permet d'énumérer les solutions d'une CNF en « bloquant » chaque solution trouvée à l'aide d'une clause. Cette approche n'est généralement pas envisageable sur des instances issues de problèmes réels, car il faut ajouter autant de clauses que de modèles, et chaque clause contient l'ensemble des variables du problème (voir [19] par exemple pour plus de détails). Cependant, cette approche fonctionne relativement bien sur les problèmes jouets que nous utilisons en enseignement.

L'architecture de Sat4j a évolué récemment pour lui permettre à terme d'intégrer un prouveur SMT. Parmi les évolutions, on retrouve la possibilité d'ajouter une clause conflictuelle à certains moments clés de la recherche (lorsqu'une solution a été trouvée par exemple). Afin de tester cette fonctionnalité, nous avons décidé de réaliser un énumérateur de solutions basé sur cette production de clause à la volée. Cette nouvelle approche permet d'énumérer les 14200 solutions d'un problème de 12 reines en 113 secondes sur un MacBook 2009 contre 593 secondes pour l'ancienne approche.

Nous avons évalué les deux approches sur les instances SAT utilisées évaluer une nouvelle technique d'énumération dans [19]. La principale difficulté était de trouver des instances satisfiables avec un nombre raisonnable de solutions (de quelques centaines à quelques centaines de millier). CBS correspond à des instances dont la taille du backbone est fixe pour des instances 3-SAT à 100 variables booléennes [25] (nous avons utilisé les plus grandes instances contenant 449 clauses). UF correspond aux instances satisfiables 3-SAT aléatoires de SATLIB (de 20 à 250 variables booléennes). FLAT correspond à des problèmes de coloriage de graphes 3-coloriables (de 20 à 600 variables booléennes). Le tableau 1 résume les résultats de cette expérimentation, effectuée sur des machines à base de processeurs Intel Quad-core XEON X5550 2,66 GHz équipées de 32 Go de mémoire avec un temps limite de 2 min. Pour chaque approche, on comptabilise comme résolue une instance pour laquelle le solveur a pu énumérer toutes les solutions de la CNF dans le temps limite.

Dans les trois cas, l'approche par ajout de clauses paresseuses permet d'obtenir de bien meilleurs résultats que l'approche par clause bloquante. Si on considère les temps de calcul sur les instances résolues par les deux approches (voir le tableau 2), on se rend compte qu'en moyenne l'approche « à la volée » énumère les solutions environ dix fois plus rapidement. Les médianes relativement proches nous apprennent que plus les instances sont difficiles, plus le gain est important, ce qui semble naturel. Par ailleurs, nous avons lancé les mêmes expérimentations, avec un temps limite de 20 minutes : bien que les deux solveurs profitent de l'augmentation du temps qui leur est imparti (voir le tableau 4), la différence d'efficacité entre les deux approches est encore plus flagrante, comme le montre le tableau 4 ; ce qui confirme que plus le nombre de modèles est important, plus le gain de l'approche « à la volée » est important dans ce contexte.

Pour conclure, lors de nos expérimentations, l'approche « à la volée » a été capable d'énumérer jusqu'à 735847 solutions en deux minutes, quand l'autre approche n'a pas pu dépasser 67637 solutions.

catégorie	#inst.	boite noire	à la volée
CBS	5000	3622	**4724**
uf	3700	2991	**3300**
flat	1700	1058	**1232**

TABLE 1 – Comparaison boite noire vs ajout de clauses à la volée pour l'énumération de solutions en nombre d'instances dont les solutions ont toutes été énumérées — temps limite de 2 minutes

catégorie	boite noire	à la volée
CBS	10373 (770)	**1029** (**647**)
uf	3653 (**94**)	**423** (103)
flat	13168 (2009)	**1319** (**1010**)

TABLE 2 – Comparaison des temps de calcul moyens (et médian) par séries, en ms, pour l'énumération de solutions — temps limite de 2 minutes, cas des instances résolues par les deux solveurs

catégorie	#inst.	boite noire	à la volée
CBS	5000	4076	**4881**
uf	3700	3110	**3408**
flat	1700	1151	**1675**

TABLE 3 – Comparaison boite noire vs ajout de clauses à la volée pour l'énumération de solutions en nombre d'instances dont les solutions ont toutes été énumérées — temps limite de 20 minutes

catégorie	boite noire	à la volée
CBS	53239 (1045)	**1584** (**694**)
uf	18264 (**101**)	**667** (121)
flat	39730 (2447)	**1672** (**1013**)

TABLE 4 – Comparaison des temps de calcul moyens (et médian) par séries, en ms, pour l'énumération de solutions — temps limite de 20 minutes, cas des instances résolues par les deux solveurs

Ces résultats encourageants nous ont conduit à généraliser ce principe au cadre des contraintes de cardinalité et des contraintes pseudo-booléennes, dans le but d'améliorer les performances des prouveurs pseudo-booléens et MAXSAT de Sat4j.

Nous présentons tout d'abord l'approche dite « boite noire » utilisée par de nombreux prouveurs (dont Sat4j) pour résoudre des problèmes d'optimisation en variables booléennes. Nous présentons ensuite l'approche « boite grise », qui permet une plus forte communication avec le prouveur SAT, pour permettre la génération de contraintes à la volée.

3 Optimisation par boite noire

Soit V un ensemble de variables booléennes. Un littéral l représente une variable booléenne $v \in V$ ou sa négation $\neg v$. On note L l'ensemble des littéraux construits à partir de V. On note $\bar{l}$ l'opposé d'un littéral, c'est à dire $\bar{l} = v$ si $l = \neg v$ et $\bar{l} = \neg v$ si $l = v$. Une clause est une disjonction de littéraux, $\bigvee l_i$. Une contrainte de cardinalité est une fonction booléenne de la forme $\sum l_i \bowtie k$ où les l_i sont des littéraux, $\bowtie \in \{<, \leq, =, \geq, >\}$ et k, le degré de la contrainte, est un entier naturel. Les variables booléennes sont interprétées comme des variables entières dans $\{0, 1\}$ avec $\bar{l} = 1 - l$. Une clause est une contrainte de cardinalité de degré 1, i.e. $a \vee \neg b \vee c \equiv a + \neg b + c \geq 1$. Toute contrainte de cardinalité peut être représentée sous la forme $\sum l_i \geq k$: $a + \neg b + c < 2 \equiv a + \neg b + c \leq 1 \equiv (1 - \neg a) + (1 - b) + (1 - \neg c) \leq 1 \equiv -\neg a - b - \neg c \leq -2 \equiv \neg a + b + \neg c \geq 2$. Une contrainte pseudo-booléenne est une fonction booléenne de la forme $\sum w_i \times l_i \bowtie k$ où les l_i sont des littéraux, $\bowtie \in \{<, \leq, =, \geq, >\}$ et les w_i, les coefficients des littéraux, et k, le degré de la contrainte, sont des entiers naturels. Une contrainte de cardinalité est une contrainte pseudo-booléenne particulière dont les coefficients sont tous égaux à 1. Toute contrainte pseudo-booléenne peut être représentée sous la forme $\sum w_i \times l_i \geq k$. Voir [22] pour plus de détails.

Soit ϕ une conjonction de contraintes booléennes (clauses, contraintes de cardinalité, contraintes pseudo-booléennes) sur l'ensemble de variables V et f

Algorithme 1 : Optimisation par renforcement (recherche linéaire)

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, et une fonction d'objectif à minimiser f

sortie : un modèle de ϕ, ou UNSAT si le problème est insatisfiable.

```
⟨réponse,certificat⟩ ← estSatisfiable (φ);
si réponse est UNSAT alors
    retourner UNSAT
fin
répéter
    m ← certificat;
    ⟨réponse,certificat⟩ ← estSatisfiable (φ ∪ {f < f (m)});
jusqu'à (réponse est UNSAT);
retourner m ;
```

une fonction linéaire de L dans $\mathbb{N}$. Il est possible de calculer un modèle de ϕ minimisant f, en utilisant l'algorithme 1. Le problème d'optimisation pseudo-booléen est alors remplacé par un nombre fini de problèmes de décision pseudo-booléens. On distingue deux approches principales pour l'implémentation de cet algorithme : celle qui réutilise des solveurs capables de raisonner avec des contraintes pseudo-booléennes nativement (des prouveurs pseudo-booléens) et celle qui réutilise des prouveurs SAT classiques et traduit la contrainte pseudo-booléenne $f < f(m)$ en CNF.

La première approche est utilisée dans Sat4j. Son principal intérêt est de représenter de manière compacte ces contraintes. Sat4j traduit les problèmes MaxSat en problèmes d'optimisation pseudo-booléens. Les résultats de Sat4j lors de la compétition MaxSat 2009 [1] le situent dans l'état de l'art, malgré des performances sur la résolution de contraintes clausales (compétition SAT [2]) bien en deça des autres prouveurs (dues principalement au choix du langage utilisé et au support de contraintes génériques).

La seconde approche est utilisée par QMaxSat[12], l'un des meilleurs prouveurs MaxSat lors des récentes compétitions MaxSat (2010-2012), en traduisant les contraintes de cardinalité (il ne gère pas les poids) sous forme de CNF [3]. QMaxSat utilise les prouveurs SAT comme des boites noires : en 2012, la version utilisant Glucose 2 s'avère plus efficace que la version utilisant Minisat sur les instances dites « industrielles » avec contraintes dures (*Industrial Partial Max Sat*), alors que la version utilisant Minisat s'avère plus efficace sur les instances dites « fabriquées » avec contraintes dures (*Crafted Partial Max Sat*). Cela montre l'intérêt de l'approche boite noire : il suffit de choisir parmi les divers solveurs disponibles celui qui fonctionne le mieux sur le type de problème à résoudre.

Une autre approche permettant la réutilisation des solveurs SAT pour résoudre des problèmes d'optimisation existe, basée elle aussi sur la traduction de contraintes pseudo-booléennes en CNF : l'optimisation basée sur la détection de noyau incohérent (*unsat core guided MaxSat solvers*)[10, 18, 2]. Cette approche est spécifique à la résolution de problèmes MaxSat, mais permet de résoudre d'autres problèmes d'optimisation par traduction vers MaxSat.

Les approches basées sur la réutilisation de prouveurs SAT ne sont pas toujours les plus adaptées. Les approches de type *Branch & Bound* fonctionnent généralement très bien sur les instances MaxSat aléatoires ou fabriquées : akmaxsat[14] était le meilleur solveur MaxSat dans ces catégories en 2011 et 2012.

4 Optimisation par contraintes de bornes dynamiques

Afin de comprendre les enjeux de l'ajout de contraintes à la volée dans un prouveur CDCL, nous devons tout d'abord présenter brièvement et informelement le fonctionnement de ce type de solveur. Nous renvoyons le lecteur à [23] pour de plus amples informations sur ce sujet.

4.1 Fonctionnement des solveurs CDCL

L'algorithme 2 représente de manière abstraite un prouveur SAT de type « Conflict Driven Clause Learning » (CDCL). Les composants principaux sont la procédure de propagation unitaire (`propager`), la procédure d'analyse de conflits (`analyser`) et l'heuristique de choix de variable (`décider`). Lors de l'apparition d'un conflit (clause falsifiée), le mécanisme de retour arrière est nécessaire pour permettre au solveur de défaire le dernier choix de l'heuristique. En effet, pour chaque conflit rencontré, une nouvelle clause c, falsifiée dans le contexte courant, est retournée par la fonction `analyser` et ajoutée au problème. Plus précisément, cette nouvelle clause est construite de telle sorte qu'elle propage une valeur de vérité quand la dernière décision est annulée car elle ne contient qu'un seul littéral affecté au dernier niveau de décision (Unique Implication Point)[24]. Ainsi, l'apprentissage de la clause devenue unitaire va permettre à la procédure `propager`

1. `http://maxsat.ia.udl.cat:81/09/`
2. `http://www.satcompetition.org/2009/`
3. De nombreux travaux concernent la traduction des contraintes de cardinalité et des contraintes pseudo-booléennes en CNF (voir par exemple [3, 1] pour un aperçu des travaux récents).

de propager une nouvelle valeur de vérité. En pratique, on ne défait pas uniquement la dernière décision, mais toutes les décisions qui n'affectent pas la clause apprise (*backjump*). On peut voir cette action comme le fait de remettre le prouveur SAT dans un état compatible avec celui d'un prouveur qui aurait eu cette clause dès le départ : la clause unitaire est propagée dès qu'elle est détectée.

Algorithme 2 : Algorithme CDCL

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ
sortie : un modèle de ϕ, ou UNSAT si le problème est insatisfiable.

```
1  répéter
2  |  conflit ← propager() ;
3  |  % invariant : les conflits sont traités dès qu'ils
   |  sont détectés ;
4  |  si conflit ≠ NIL alors
5  |  |  si pileDecisionsVide() alors
6  |  |  |  retourner UNSAT;
7  |  |  fin
8  |  |  clause ← analyser (conflit);
9  |  |  défaireChoixPourPropager (clause);
10 |  |  apprendre (clause);
11 |  sinon
12 |  |  si décider() = NIL alors
13 |  |  |  retourner SAT;
14 |  |  fin
15 |  fin
16 jusqu'à temps limite atteint;
17 retourner TIMEOUT;
```

4.2 Ajouter des contraintes à la volée

Lorsque l'on souhaite ajouter des contraintes de borne à la volée dans le solveur, on se trouve dans une situation différente de celle de l'apprentissage de clauses : la contrainte ajoutée est falsifiée, mais la décision ayant amené cette satisfaction n'est pas nécessairement la dernière à avoir été prise [4]. Il est donc nécessaire de dépiler toutes les propagations et décisions jusqu'au premier niveau où la contrainte est falsifiée de manière à ce que l'état du solveur soit cohérent avec l'approche CDCL : un conflit doit être traité dès qu'il est détecté. L'algorithme 3 représente un solveur de type CDCL capable de travailler avec différents type de contraintes, avec ajout de contraintes à la volée lorsqu'une solution est trouvée. On note ligne 11 une fonction `defaireJusqueFalsif` qui a pour but de dépiler les décisions jusqu'au niveau de celle provoquant la falsification de la contrainte. Cette fonction agit de manière spécifique pour chaque type de contraintes :

clause une clause est falsifiée ssi tous ses littéraux sont falsifiés, il suffit donc de dépiler toutes les décisions jusqu'au dernier niveau de décision de la clause. On se retrouve ici dans un cas similaire à un retour arrière lors de l'apprentissage d'une clause après analyse de conflit.

cardinalité une contrainte de cardinalité de n littéraux et de degré k est falsifiée ssi $n-k+1$ littéraux sont falsifiés. Il faut donc détecter les $n-k+1$ premières falsifications de littéraux.

pseudo-booléenne une contrainte pseudo-boléenne de n littéraux et de degré k est falsifiée ssi $\sum_{i=1}^{n} w_i \times l_i \equiv \sum w_i - \sum_{l_j=0} w_j < k$. Il faut donc détecter le premier niveau de décision pour lequel la somme des poids des littéraux falsifiés dépasse $\sum w_i - k$.

On peut remarquer que dans le cas des contraintes de cardinalité et des contraintes pseudo-booléennes, l'ordre dans lequel les littéraux sont falsifiés est important, ce qui n'est pas le cas pour les clauses.

Notons enfin que les solveurs CDCL modernes effacent les clauses apprises durant la recherche. Il est donc nécessaire que les contraintes de borne soient gérées spécifiquement, pour éviter qu'elles ne soient effacées. On notera de plus qu'il n'est pas nécessaire de garder plus d'une contrainte de borne, car ces contraintes sont de plus en fortes (c'est aussi le cas dans l'approche par boite noire [5]).

Si maintenir l'invariant que chaque conflit est analysé dès qu'il est détecté est obligatoire pour garantir le bon fonctionnement d'un prouveur CDCL lors de l'ajout de contraintes à la volée, nous allons aussi voir qu'en fonction du type de contrainte ajouté certaines vérifications doivent être faîtes afin de garantir l'efficacité du solveur.

4.3 Propager les contraintes ajoutées à la volée

Il est possible que la contrainte ajoutée puisse propager des valeurs de vérité lorsque l'on défait des décisions après l'analyse de conflits.

Considérons par exemple la contrainte $l_1 + l_2 + ... + l_n \leq 0$ produite comme contrainte de borne pour un modèle qui satisfait un unique littéral (l_j) de la fonction objectif. Après ajout de cette contrainte, tous les littéraux qu'elle contient doivent être propagés à faux au niveau de décision 0, puisque qu'un modèle doit dorénavant contenir $\neg l_1$, $\neg l_2$, ..., et $\neg l_n$. En revanche, si on considère uniquement la clause issue de l'analyse de conflits, nous ne pouvons propager que

4. Ce qui était le cas dans notre étude préliminaire d'énumération de modèles, car la clause contenait toutes les variables.

Algorithme 3 : Optimisation par ajout de contraintes de bornes dynamique

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, et une fonction d'objectif à minimiser f

sortie : un modèle de ϕ minimisant f, ou UNSAT si le problème est insatisfiable.

```
1  satisfiable ← FALSE;
2  dynConflit ← NIL;
3  répéter
4      conflit ← propager ();
5      si conflit = NIL alors
6          si décider() = NIL alors
7              dynConflit ← genContrainteBorne() ;
8              si dynConflit = NIL alors
9                  retourner SAT;
10             fin
11             défaireJusqueFalsif (dynConflit);
12             satisfiable ← TRUE;
13             conflit ← dynConflit ;
14         fin
15     fin
16     si conflit ≠ NIL alors
17         si pileDecisionsVide() alors
18             si satisfiable alors
19                 retourner OPT;
20             sinon
21                 retourner UNSAT;
22             fin
23         fin
24         clause ← analyser (conflit);
25         défaireChoixPourPropager (clause);
26         si conflit = dynConflit alors
27             propager (dynConflit);
28             dynConflit ← NIL;
29         fin
30         apprendre (clause);
31     fin
32 jusqu'à temps limite atteint;
33 retourner TIMEOUT ;
```

le littéral $\neg l_j$, et non les littéraux l_i tels que $i \neq j$ ($i \in \{1, 2, ..., n\}$). Il est donc nécessaire de permettre à la nouvelle contrainte de propager des valeurs de vérité. C'est le but de la fonction `propager` ligne 27 dans l'algorithme 3.

Il existe de plus un problème spécifique aux contraintes pseudo-booléennes : celles-ci ont la particularité de pouvoir propager des valeurs à divers niveaux de décision, alors qu'une clause ne propage qu'une valeur à un unique niveau de décision, et qu'une contrainte de cardinalité peut propager plusieurs valeurs de vérité là aussi à un unique niveau de décision. L'ajout de contraintes pseudo-booléennes implique donc de remonter au premier niveau de propagation, qui peut être le niveau de propagation 0 si le poids d'un littéral est nécessaire à la satisfaction de la contrainte : $5x_1 + 3x_2 + x_3 \geq 6$ par exemple implique x_1 au niveau de décision 0. Cette contrainte peut être falsifiée soit en falsifiant x_1, soit en falsifiant x_2 et x_3. Dans le premier cas, la propagation se fera effectivement au niveau de décision 0. Dans le second, x_1 ne sera pas propagé. Nous n'avons pas trouvé de solution élégante pour régler ce problème à l'heure actuelle.

Dans le cas où la contrainte est une simple clause, une seule valeur de vérité peut être propagée (le seul littéral non falsifié), et cette valeur sera propagée par la clause issue de l'analyse de conflits. En effet, si une clause apprise propage une valeur de vérité quand les décisions sont dépilées, c'est qu'il s'agit d'une clause contenant un seul littéral au niveau de décision courant (*First UIP*). Donc la procédure d'analyse de conflits va retourner exactement cette clause, ou une clause simplifiée [26] dont le littéral propagé sera exactement celui qui serait propagé par la clause ajoutée à la volée. Il n'y a donc pas de propagation supplémentaire possible lors de la génération de clauses conflictuelles à la volée.

5 Expérimentations et résultats

Nous avons intégré notre algorithme d'optimisation par contrainte de bornes dynamiques à la plateforme Sat4j, dans les solveurs d'optimisation pseudo-booléenne et MaxSat. Il est à noter qu'aucun de ces solveurs n'effectue de pré-traitement. Nos expérimentations ont été menées sur des machines équipées de processeurs Intel XEON 3,0 GHz associées à 2 Go de mémoire.

En plus des résultats de performance globale, nous souhaitons vérifier les hypothèses concernant les cas ou une approche par contraintes de borne dynamique peut s'avérer plus intéressante que lors d'une approche boite noire.

– La preuve unsat finale devrait être plus courte,

puisque l'on bénéficie de l'analyse de conflits
– Mais dans certains cas, on risque d'énumérer énormément de solutions

Voyons si ces hypothèses se vérifient en pratique.

5.1 Optimisation pseudo-booléenne

Nous avons utilisé des benchmarks issus de la compétition PB10 [5] afin de comparer les résultats de nos deux approches, en laissant un temps de trente minutes à nos solveurs pour trouver une solution optimale (soit le même temps limite que celui de la compétition). Le tableau 5 présente le nombre d'instances résolues par nos deux algorithmes (le nombre d'instances prouvées insatisfiables est indiqué entre parenthèses).

catégorie	#inst.	b. noire	b. grise
BIGINT-LIN	532	**125** (57)	115 (57)
SMALLINT-LIN	699	**270** (33)	266 (33)
SMALLINT-NLC	409	273 (0)	**275** (0)

TABLE 5 – Instances PB10 résolues (et UNSAT)

Concernant ces benchmarks, on remarque un léger avantage à l'utilisation de l'algorithme utilisant le solveur SAT en tant que boîte noire. Cependant, ces chiffres sont à nuancer légèrement car ces résultats globaux cachent le fait que chaque approche est capable de résoudre des problèmes que l'autre approche ne peut pas résoudre : 12 instances sont uniquement résolues par l'approche externe, et 6 instances sont uniquement résolues par l'approche interne. Le cas pathologique de l'approche à base de contraintes de bornes dynamiques est d'énumérer un nombre de solutions beaucoup plus important que l'approche boite noire, ce qui fait perdre du temps, mais aussi beaucoup de mémoire. C'est par exemple le cas pour l'instance `normalized-mps-v2-20-10-glass4.opb`, pour laquelle le solveur « boîte noire » trouve la solution optimale au bout de 50 secondes en ayant énuméré 16 solutions intermédiaires, alors que le solveur utilisant l'autre approche meurt car il nécessite trop de mémoire, après avoir calculé 8715 solutions intermédiaires. Ce cas pathologique n'est présent que rarement dans nos expérimentations, bien que cette dernière approche a tendance à énumérer plus de solutions avant de trouver la valeur optimale, comme le montre la figure 1. En termes quantitatifs, l'algorithme d'optimisation classique énumère en moyenne 5 solutions quand l'algorithme utilisant des contraintes de borne dynamiques énumère en moyenne 17 solutions. Le nombre médian de solutions énumérées est lui de 2 dans les deux cas, ce qui montre que la différence entre les nombres de solutions intermédiaires parcourue par nos algorithmes est d'autant plus importante que le nombre de solutions énumérées est important.

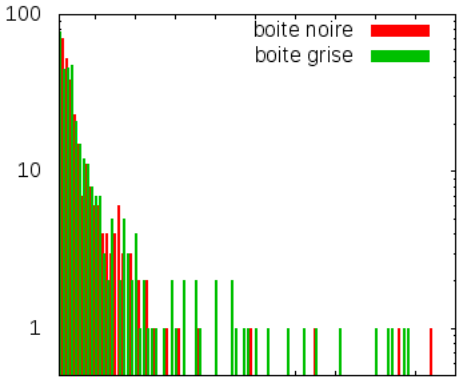

FIGURE 1 – Nombre de solutions énumérées pour PB (graphe partiel jusqu'à 100 solutions) — nombre de solutions énumérées par instances en abscisse, nombre d'instances en ordonnée

La figure 2 représente la distribution des temps des solveurs sous forme de « cactus », c'est à dire en triant les temps par ordre croissant. Les prouveurs peuvent avoir des temps différents sur des instances individuelles, mais globalement, la distribution des temps est quasi-identique dans les deux cas. La figure 3 représente de la même manière la distribution du temps nécessaire à la preuve de l'optimalité, c'est à dire à la résolution du dernier problème de décision, non satisfiable. La encore, si les temps individuels par instance varient, les distributions de temps ne sont pas dissociables.

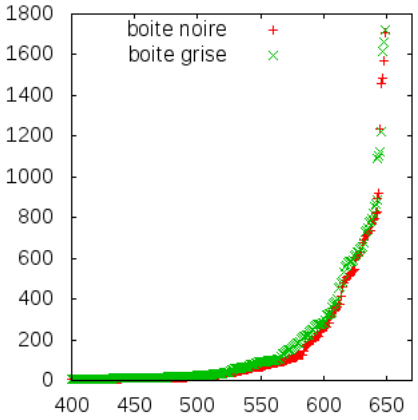

FIGURE 2 – Temps de résolution des instances PB

5. `http://www.cril.univ-artois.fr/PB10/`

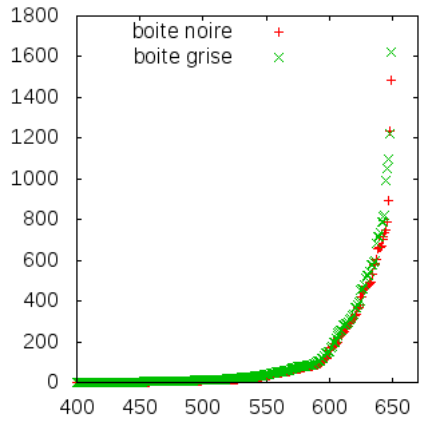

FIGURE 3 – Temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée — cas de PB

5.2 Optimisation MaxSat

Nous avons comparé ces deux approches dans le cadre de l'optimisation MaxSat sur les instances de la compétition MaxSat10, avec un temps limite de vingt minutes (soit le même que lors de la compétition). Les résultats sont présentés dans le tableau 6.

catégorie	#inst.	b. noire	b. grise
ms_crafted	167	**2**	**2**
ms_industrial	77	**8**	5
ms_random	300	**0**	**0**
pms_crafted	385	**190**	187
pms_industrial	497	**270**	258
pms_random	240	**26**	**26**
wms_crafted	149	**43**	**43**
wms_random	200	**16**	**16**
wpms_crafted	378	**146**	142
wpms_industrial	132	**36**	35
wpms_random	150	**29**	**29**

TABLE 6 – Instances MaxSat10 résolues

Ici encore, les résultats sont très proches, avec un très léger avantage pour l'approches « boîte noire ». Encore une fois, cette légère différence est en partie imputable à une énumération d'un trop grand nombre de solutions concernant l'autre algorithme pour certaines instances. Dans ces jeux de test, 31 instances sont résolues par l'approche externe et non par l'interne, et 8 autres sont dans le cas inverse. De la même manière que l'algorithme d'optimisation par contraintes de bornes dynamiques énumère de manière générale plus de solutions dans le cadre de l'optimisation pseudo-booléenne, on observe un comportement similaire pour MaxSat (voir la figure 4). En termes de statistiques, l'approche « boîte noire » énumère en moyenne 7 solutions quand l'approche « boîte grise » en énumère 13. Encore une fois, une étude du nombre médian de modèles découverts lors de la recherche (4 pour la première, 5 pour la deuxième) nous montre que plus le nombre de solutions énumérées est important, plus la différence entre les deux approches sur ce point l'est aussi.

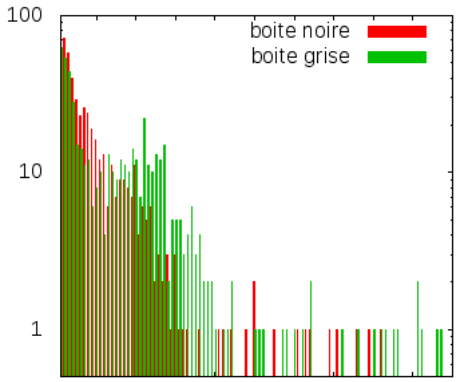

FIGURE 4 – Nombre de solutions énumérées pour MaxSat (graphe partiel jusqu'à 100 solutions) — nombre de solutions énumérées par instances en abscisse, nombre d'instances en ordonnée

Une fois encore, comme le montrent les figures 5 et 6, on ne peut départager de manière globale nos deux approches ni sur la distribution des temps de calcul des solutions optimales, ni sur la distribution des temps de la preuve d'optimalité.

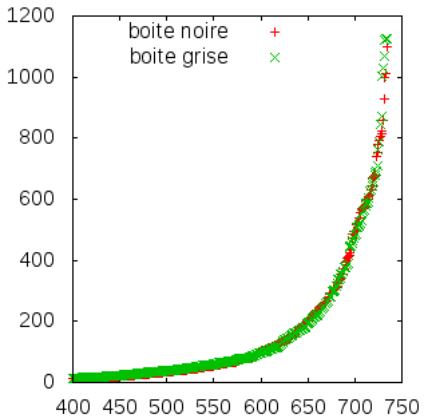

FIGURE 5 – Temps de résolution des instances MaxSat

6 Analyse des résultats

A la lumière de ces expérimentations, l'approche par apprentissage de bornes à la volée ne s'avère pas plus efficace que l'approche incrémentale ou boite noire.

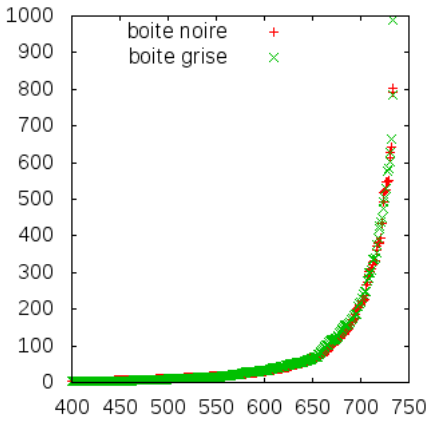

FIGURE 6 – Temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée — cas de MaxSat

Ces résultats sont décevants compte tenu des résultats initiaux obtenus sur l'énumération de solutions. Il nous semble important de mettre en avant quelques faits qui peuvent expliquer ces résultats.

Tout d'abord, dans nos deux problèmes d'optimisation, les prouveurs trouvent une solution très proche de l'optimale dans plus de la moitié des cas (nombre de solutions intermédaires médian variant de 2 à 5 selon les problèmes et les approches). Il faut mettre cela en balance avec les milliers de solutions générées dans le cadre de l'énumération.

Les cas (peu nombreux) pour lesquels une solution n'est pas trouvée sont souvent liés à une grande différence en terme de nombre de solutions/contraintes générées. En énumération de solutions, les deux approches doivent générer exactement le même nombre de solutions. Il n'est cependant pas clair à l'heure actuelle si cela est dû à l'approche d'ajout de contraintes à la volée, qui force le solveur à continuer de chercher des solutions dans l'espace de recherche courant ou s'il s'agit d'illustrations de la faiblesse de propagation mentionnée dans la section précédente.

Enfin, nous avons essayé différentes heuristiques, stratégies de redémarrages, stratégies de minimisation de clauses : les résultats individuels des solveurs changent légèrement, mais les tendances restent identiques.

7 Conclusions et perspectives

Nous avons présenté dans le papier les résultats de notre étude sur l'utilisation d'une communication privilégiée avec un prouveur SAT générique permettant l'ajout de contraintes à la volée pour résoudre des problèmes d'optimisation en variables booléennes. Nous avons montré que si l'ajout dynamique de clauses se fait sans difficulté, l'ajout de contraintes de cardinalité ou de contraintes pseudo-booléennes nécessite quelques précautions. Les résultats expérimentaux sont assez décevants : on ne remarque pas de différence flagrante d'efficacité, comme on avait pu le constater sur le cas de l'énumération de solutions qui avait motivé cette étude. Ces résultats peuvent s'expliquer par la nature des benchmarks utilisés : nombre d'entre eux sont résolus après production de quelques solutions intermédiaires. Les solveurs SAT utilisant des stratégies de redémarrage rapide, et bénéficiant d'une interface incrémentale, il semble que le gain d'une analyse de conflit sur les contraintes ajoutées à la volée soit quasi nul. Nous avons identifié une faiblesse dans notre approche lors de l'apprentissage de contraintes pseudo-booléennes, qui pourrait manquer certaines propagations. Nous allons vérifier si cela se produit effectivement en pratique et chercher un moyen d'éviter ce problème.

Références

[1] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *J. Artif. Intell. Res. (JAIR)*, 45 :443–480, 2012.

[2] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artif. Intell.*, 196 :77–105, 2013.

[3] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks : a theoretical and empirical study. *Constraints*, 16(2) :195–221, 2011.

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [7], pages 825–885.

[5] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3) :59–6, 2010.

[6] Armin Biere. Bounded model checking. In Biere et al. [7], pages 457–481.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[8] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[9] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4) :543–560, 2003.

[10] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

[11] Ian P. Gent. Arc consistency in sat. In Frank van Harmelen, editor, *ECAI*, pages 121–125. IOS Press, 2002.

[12] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat : A partial max-sat solver. *JSAT*, 8(1/2) :95–100, 2012.

[13] Daniel Kroening. Software verification. In Biere et al. [7], pages 505–532.

[14] Adrian Kuegel. Improved exact solver for the weighted max-sat problem. In *Workshop Pragmatics of SAT*, 2010.

[15] Chu Min Li and Felip Manyà. Maxsat, hard and soft constraints. In Biere et al. [7], pages 613–631.

[16] Vasco M. Manquinho and João P. Marques Silva. On solving boolean optimization with satisfiability-based algorithms. In *AMAI*, 2000.

[17] Sylvain Merchez, Christophe Lecoutre, and Frédéric Boussemart. Abscon : A prototype to solve csps with abstraction. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 730–744. Springer, 2001.

[18] António Morgado, Federico Heras, and João Marques-Silva. Improvements to core-guided binary search for maxsat. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2012.

[19] António Morgado and João P. Marques Silva. Good learning and implicit model enumeration. In *ICTAI*, pages 131–136. IEEE Computer Society, 2005.

[20] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3) :357–391, 2009.

[21] Jussi Rintanen. Planning and sat. In Biere et al. [7], pages 483–504.

[22] Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In Biere et al. [7], pages 695–733.

[23] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Biere et al. [7], pages 131–153.

[24] João P. Marques Silva and Karem A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5) :506–521, 1999.

[25] Josh Singer, Ian P. Gent, and Alan Smaill. Backbone fragility and the local search cost peak. *J. Artif. Intell. Res. (JAIR)*, 12 :235–270, 2000.

[26] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[27] Toby Walsh. Sat v csp. In Rina Dechter, editor, *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.