Coopération de solveurs linéaires sur les réels pour la résolution de problèmes linéaires en nombres entiers

Christine Solnon

LISI - Université Claude Bernard et INSA de Lyon
43, bd du 11 novembre 1918 - Bat. 710
69 622 Villeurbanne Cedex - France
E-mail : `csolnon@bat710.univ-lyon1.fr`

Résumé **:** *Pour résoudre des problèmes linéaires en nombres entiers, une approche largement utilisée consiste à effectuer une recherche globale par séparation et évaluation à travers toutes les combinaisons possibles. Du fait de l'aspect fortement combinatoire du problème, cette recherche est contrôlée par un solveur de contraintes qui utilise les contraintes pour réduire a priori l'espace de recherche.*

On introduit dans cet article un nouveau solveur basé sur la coopération de solveurs linéaires sur les réels. L'idée est d'utiliser un ensemble de solveurs linéaires pour calculer les domaines des variables sur les réels, puis de réduire ces domaines à l'intervalle entier inclus le plus proche. Ce filtrage est effectué jusqu'à atteindre un point fixe où tous les domaines sont bornés par des entiers appartenant à l'espace des solutions sur les réels. Ce point fixe correspond à une nouvelle consistance partielle qui réduit fortement l'espace de recherche et permet de résoudre des PLNEs plus efficacement.

Mots-clés **:** *Problèmes linéaires en nombres entiers, Programmation par contraintes, Coopération de solveurs*

---

1 Introduction

Les problèmes linéaires en nombres entiers — appelés PLNEs — ont de nombreuses applications industrielles telles que l'allocation de ressources, la planification ou encore l'ordonnancement de tâches. Dans le cas général, un PLNE se ramène au problème de la figure 1.

Dans un PLNE, certaines variables (celles qui appartiennent à X_{int}) sont contraintes à prendre des valeurs entières. Le problème obtenu en relaxant

$$
\begin{array}{lll}
\text{minimiser} & \sum_{j=1}^{n} c_j x_j & \\
\text{tel que} & \sum_{j=1}^{n} a_{ij} x_j \ \leq \ b_i & \forall i \in 1..m \\
 & \sum_{j=1}^{n} a_{ij} x_j \ = \ b_i & \forall i \in m+1..p \\
 & l_j \leq x_j \leq u_j & \forall j \in 1..n \\
 & x_j \in Z & \forall x_j \in X_{int}
\end{array}
$$

Figure 1 : *PLNE sous forme standard*

ces contraintes d'intégralité est appelé *relaxation continue* du PLNE et est facilement soluble (par exemple, en utilisant l'algorithme du Simplexe ou du point intérieur). En revanche, la prise en compte des contraintes d'intégralité rend le problème difficile et intrinsèquement combinatoire (NP-complet).

Une approche largement utilisée pour résoudre un PLNE consiste à effectuer une recherche globale et complète par séparation et évaluation ("branch & bound") de la façon suivante [JNS97] :

1 - résoudre une relaxation du PLNE
2 - **si** la relaxation est inconsistante,
3 - **alors** le PLNE n'a pas de solution
4 - **sinon si** toutes les contraintes relaxées sont aussi satisfaites,
5 -   **alors** le PLNE est résolu
6 -   **sinon** séparer le PLNE en plusieurs sous-PLNEs ;
7 -   réitérer le processus en 1 sur chacun des sous-PLNEs

La phase de relaxation est effectuée à chaque point de choix de la recherche. Elle consiste à relaxer certaines contraintes du PLNE pour vérifier partiellement sa consistance. L'objectif de cette phase est de détecter le plus tôt possible dans la recherche les inconsistances et de réduire l'espace de recherche.

Dans le cas de PLNEs, la phase de résolution de la relaxation est généralement effectuée par un solveur linéaire sur les réels, qui vérifie la consistance de la relaxation continue du problème. Un solveur linéaire sur les réels est fondé sur l'algorithme du Simplexe[1] et permet de résoudre efficacement des problèmes d'optimisation linéaire sur les réels ou les rationnels [Chv83]. En

1. L'algorithme du point intérieur est globalement plus efficace que l'algorithme du Simplexe pour résoudre de gros problèmes linéaires sur les réels. Néanmoins, cet algorithme est moins bien adapté à la prise en compte incrémentale de nouvelles contraintes et n'est donc généralement pas utilisé dans une procédure de recherche par séparation et évaluation.

revanche, un solveur linéaire sur les réels ne permet pas de prendre en compte les contraintes d'intégralité sur les variables. Ces contraintes ne sont vérifiées qu'a posteriori, lorsque la recherche tente d'affecter une valeur entière à une variable et, par conséquent, de nombreuses branches de l'arbre de recherche sont fastidieusement explorées alors que les domaines de certaines variables entières ne contiennent plus de valeur entière.

Afin de prendre en compte partiellement ces contraintes d'intégralité, et réduire l'espace de recherche, deux techniques de propagation locale ont essentiellement été proposées :

- les techniques de pré-traitement et d'investigation[2], plus particulièrement utilisées dans la communauté "recherche opérationnelle" pour résoudre des PLNEs [Sav94, JNS97] ;
- les techniques de filtrage de domaines, plus particulièrement utilisées dans la communauté "intelligence artifielle" pour résoudre des problèmes de satisfaction de contraintes [Hen95, Ben95].

1.1 *Techniques locales de pré-traitement et d'investigation*

L'idée des techniques locales de pré-traitement est de considérer successivement chaque contrainte, conjointement avec les bornes des domaines de ses variables, afin de

- supprimer la contrainte si elle est redondante,
- détecter une inconsistance si la contrainte n'est pas satisfiable, ou
- réduire le domaine d'une variable si la contrainte devient insatisfiable quand la variable est fixée à une de ses bornes.

Les techniques d'investigation permettent d'anticiper les effets de ce prétraitement d'une étape en avant. L'idée est de fixer temporairement une variable à une de ses bornes avant d'effectuer à nouveau les techniques de prétraitement. Si le problème devient inconsistant, alors le domaine de la variable peut être réduit ; si d'autres variables ont leurs domaines restreints, alors une nouvelle contrainte qui renforce la relaxation continue peut être dérivée ; si une contrainte devient redondante, alors sa formulation peut être renforcée en réduisant ses coefficients. Ces techniques d'investigation ne sont généralement mises en œuvre que sur les variables binaires (dont le domaine est $\{0,1\}$).

2. "preprocessing and probing techniques" en anglais

Ces processus de pré-traitement et d'investigation sont répétés jusqu'à ce que plus aucune contrainte ne soit ajoutée. Le point fixe ainsi obtenu correspond à une consistance partielle qui prend en compte certaines contraintes d'intégralité et permet ainsi de renforcer la relaxation continue du problème.

Ces techniques de pré-traitement et d'investigation ont notamment été implémentées dans MINTO (Mixed INTeger Optimizer) [SN93]. Les expérimentations montrent que ces techniques peuvent considérablement réduire l'espace de recherche et améliorer l'efficacité de la résolution sur de nombreux PLNEs. Néanmoins, sur certains PLNEs, ces techniques ne réduisent que faiblement le nombre de points de choix et les performances en temps sont alors relativement dégradées [JNS97].

1.2 *Techniques locales de filtrage de domaines*

L'idée du filtrage de domaines est d'utiliser successivement chaque contrainte pour réduire les domaines de ses variables jusqu'à ce qu'aucune réduction ne puisse plus être effectuée. Ce point fixe correspond à une certaine consistance partielle, généralement une approximation de la consistance d'arc (consistance de boite [BMH94] ou 2B-consistance [Lho93]).

Les solveurs par filtrage de domaines sont très largement utilisés pour résoudre des problèmes de satisfaction de contraintes (non nécessairement linéaires) sur les domaines finis (par exemple, dans CHIP [DHS+88], cc(FD) [HSD95] et clp(FD) [Dia95]) ou encore sur les réels (par exemple, dans Newton [BMH94], Interlog [Lho93] et Prolog IV [Pro96]).

Berenger et De Backer ont intégré dans [BB94, BB96] un solveur par filtrage de domaines avec un solveur linéaire sur les réels. Ce solveur coopératif a été notamment utilisé pour la résolution de PLNEs : le solveur par filtrage de domaines envoie au solveur linéaire chaque nouvelle borne entière qu'il déduit, tandis que le solveur linéaire vérifie que la relaxation du problème sur les réels est consistante. Cette approche permet de réduire l'espace de recherche et de résoudre plus efficacement certains PLNEs.

1.3 *Analyse locale versus analyse globale*

Le filtrage de domaines comme le pré-traitement et l'investigation exploitent chaque contrainte localement (indépendamment des autres contraintes du problème) pour déduire de nouvelles informations. Ces vérifications locales sont relativement faciles à mettre en œuvre mais, en contrepartie, elles sont parfois très incomplètes. Considérons par exemple l'ensemble de contraintes suivant

$$x_1 - x_2 \leq 0$$
$$x_1 + x_2 \leq 1$$

où x_1 et x_2 sont des variables binaires (à valeur dans $\{0,1\}$). Si on considère chaque contrainte séparément, aucun domaine ne peut être réduit[3]. Néanmoins, la combinaison de ces deux contraintes restreint la borne supérieure de x_1 à 0.5 (en effet, si x_1 est supérieur à 0.5, alors la première contrainte impose que x_2 soit supérieur à 0.5 tandis que la seconde contrainte impose que x_2 soit inférieur à 0.5). Ainsi, une analyse globale de ce système permettrait de fixer x_1 à 0, puis de supprimer ces deux contraintes devenues redondantes.

Dans ce papier, on définit un solveur par filtrage de domaines qui *exploite de façon globale l'ensemble des contraintes linéaires.* L'idée est d'utiliser un solveur linéaire sur les réels, non pas uniquement de façon passive pour vérifier que la relaxation continue est consistante, mais également de façon active pour réduire les domaines des variables. En effet, un solveur linéaire a une vue globale de l'ensemble des contraintes linéaires et permet de calculer les domaines effectifs des variables entières par rapport à la relaxation continue du PLNE. Ces domaines sont alors restreints à l'intervalle entier inclus le plus proche. Ce filtrage est effectué jusqu'à atteindre un point fixe où tous les domaines sont bornés par des entiers qui appartiennent à l'espace des solutions de la relaxation continue du problème. Ce point fixe correspond à une nouvelle consistance partielle, appelée LIB-consistance (Linear Integer Bounding). La LIB-consistance prend en compte partiellement les contraintes d'intégralité et permet de résoudre de nombreux PLNEs plus efficacement, en nombre de points de choix mais aussi en temps.

1.4 *Plan de l'article*

Dans le paragraphe 2, on introduit la notation utilisée. Dans le paragraphe 3, on définit formellement les notions de LIB-consistance et de LIB-filtrage d'un PLNE. L'algorithme effectuant le LIB-filtrage est décrit dans le paragraphe 4. Cet algorithme utilise coopérativement un ensemble de solveurs linéaires sur les réels. L'implémentation de ces solveurs est décrite dans le paragraphe 5. On compare ensuite dans le paragraphe 6 un solveur effectuant un LIB-filtrage avec un simple solveur linéaire sur les réels pour la résolution d'un ensemble de PLNEs. Enfin, on discute dans le paragraphe 7 d'autres approches similaires et des perspectives de recherche.

3. Toutefois, des techniques d'anticipation assurant des consistances plus fortes telles que la 3B-consistance permettraient de réduire le domaine de x_1.

2 Notations

Le PLNE général de la figure 1 sera dénoté par le triplet (X_{int}, C, z) où X_{int} est l'ensemble des variables entières, C est l'ensemble des contraintes linéaires (équations et inéquations linéaires et contraintes de bornes sur les variables) et z est la fonction linéaire à minimiser :

$$
\begin{array}{rll}
C = & \{\sum_{j=1}^{n} a_{ij}x_j \ \leq \ b_i, & \forall i \in 1..m\} \\
\cup & \{\sum_{j=1}^{n} a_{ij}x_j \ = \ b_i, & \forall i \in m+1..p\} \\
\cup & \{l_j \leq x_j \leq u_j, & \forall j \in 1..n\} \\
& & \\
z = & \sum_{j=1}^{n} c_j x_j &
\end{array}
$$

Afin de simplifier la présentation de l'algorithme de LIB-filtrage, les contraintes spécifiant les bornes des variables sont intégrées dans l'ensemble des contraintes linéaires C. On notera $D_C(x_i)$ le domaine de la variable x_i défini dans un ensemble de contraintes C, de telle sorte que $D_C(x_i) = [l_i..u_i]$ si la contrainte $l_i \leq x_i \leq u_i$ appartient à C.

Etant donnée une valeur réelle v, on notera
- $\lfloor v \rfloor$ la plus grande valeur entière telle que $\lfloor v \rfloor \leq v$, et
- $\lceil v \rceil$ la plus petite valeur entière telle que $v \leq \lceil v \rceil$.

3 Définition de la LIB-consistance et du LIB-filtrage

On propose de prendre en compte partiellement les contraintes d'intégralité afin de renforcer la consistance et réduire l'espace de recherche. L'idée est de vérifier que le domaine de chaque variable entière est borné par des valeurs entières qui appartiennent à l'ensemble des solutions de la relaxation continue du PLNE. En d'autres termes, chaque facette de l'hyper-boite définie par les bornes des variables entières doit avoir une intersection non vide avec le polyèdre convexe défini par les contraintes linéaires C.

De façon plus formelle, cette consistance partielle, appelée LIB-consistance (pour Linear Integer Bounding), est définie de la façon suivante :

Définition 1 *Le PLNE (X_{int}, C, z) est LIB-consistant si pour chaque variable entière $x_j \in X_{int}$ telle que $D_C(x_j) = [l_j..u_j]$, les trois propriétés suivantes sont vérifiées :*
1- *les bornes l_j et u_j ont des valeurs entières, i.e., $l_j \in Z$ et $u_j \in Z$,*
2- *$C \cup \{x_j = l_j\}$ est consistant sur les réels,*
3- *$C \cup \{x_j = u_j\}$ est consistant sur les réels.*

Etant donné un PLNE (X_{int}, C, z), l'hyper-boite définie par les bornes des variables entières de X_{int} englobe avec une marge généralement non nulle le polyèdre défini par C, de telle sorte que le problème n'est pas LIB-consistant. Dans ce cas, on doit réduire les domaines des variables entières jusqu'à ce que le PLNE devienne LIB-consistant. Ce filtrage doit être effectué sans pour autant supprimer de solutions au problème. De façon plus formelle, un LIB-filtrage est défini de la façon suivante :

Définition 2 *Un LIB-filtrage d'un PLNE (X_{int}, C, z) est un PLNE (X_{int}, C', z) tel que :*
- pour toute variable entière $x_i \in X_{int}$, on a $D_{C'}(x_i) \subseteq D_C(x_i)$,
- (X_{int}, C', z) et (X_{int}, C, z) admettent le même ensemble de solutions entières,
- (X_{int}, C', z) est LIB-consistant.

4 Algorithme de LIB-filtrage

Un LIB-filtrage est effectué par l'algorithme de propagation — adapté de l'algorithme de Waltz [Wal72] — décrit dans la figure 2. Cet algorithme utilise, pour chaque variable entière $x_i \in X_{int}$, deux solveurs linéaires $min[x_i]$ et $max[x_i]$ respectivement chargés de calculer les valeurs minimale et maximale l_r et u_r que x_i peut prendre sur les réels[4]. Le domaine de x_i est alors restreint à l'intervalle entier immédiatement inclus, c'est à dire $\lceil l_r \rceil \leq x_i \leq \lfloor u_r \rfloor$. Ces nouvelles bornes entières sont ajoutées à C pour restreindre à nouveau d'autres domaines.

Dans cet algorithme, Q contient l'ensemble des variables entières à filtrer. Au départ, Q est initialisé avec l'ensemble X_{int} des variables entières. A chaque itération, une variable x_i est enlevée de Q. Si le domaine de x_i est réduit, alors Q est réinitialisé avec l'ensemble $X_{int} - \{x_i\}$ des variables entières dont le domaine est susceptible d'être réduit de nouveau.

Terminaison du LIB-filtrage : L'algorithme s'arrête soit quand Q est vide (autrement dit quand plus aucun domaine ne peut être réduit), soit quand l'ensemble C des contraintes devient inconsistant sur les réels. La terminaison du processus est garantie par le fait qu'il y a un nombre fini de variables, et que leurs domaines sont finis : à chaque itération, soit une variable est enlevée de l'ensemble Q des variables à filtrer et les domaines des variables ne sont pas modifiés, soit l'ensemble Q des variables à filtrer augmente mais le domaine d'une variable est réduit dans C.

4. Chaque solveur linéaire est implémenté par un Simplexe. Cette implémentation est décrite au paragraphe 5.

fonction `LIB-filtre`(C: un ensemble de contraintes)
retourne un ensemble de contraintes

  $Q \leftarrow X_{int}$
  tant que $Q \neq \emptyset$ **et** C est consistant sur les réels **faire**
    enlever de Q une variable entière x_i
    $u_r \leftarrow max[x_i].optimise(C)$
    $l_r \leftarrow min[x_i].optimise(C)$
    soit $D_C(x_i) = [l_i ... u_i]$, le domaine de x_i dans C
    si $l_i < \lceil l_r \rceil$ **ou** $\lfloor u_r \rfloor < u_i$
    alors $C \leftarrow C \cup \{\lceil l_r \rceil \leq x_i \leq \lfloor u_r \rfloor\}$
      $Q \leftarrow X_{int} - \{x_i\}$
    finsi
  fin tant que
  retourne(C)
fin `LIB-filtre`

où $min[x_i]$ et $max[x_i]$ sont 2 solveurs linéaires tels que la fonction $optimise(C)$ de $min[x_i]$ (resp. $max[x_i]$) rend la valeur réelle minimale (resp. maximale) que x_i peut prendre sous l'ensemble de contraintes C.

Figure 2 : *Algorithme de propagation de la LIB-consistance*

Proposition 3 *Le système de contraintes retourné par* `LIB-filtre(`C`)` *admet le même ensemble de solutions sur les entiers que* C.

En effet, les contraintes ajoutées à C sont de la forme $\lceil l_r \rceil \leq x_i \leq \lfloor u_r \rfloor$. Ces contraintes suppriment l'ensemble des solutions sur les réels telles que $l_r \leq x_i < \lceil l_r \rceil$ ou $\lfloor u_r \rfloor < x_i \leq u_r$. Ces deux intervalles ne comprenant pas de valeur entière, leur suppression ne coupe pas de solution sur les entiers.

Proposition 4 *Le système de contraintes retourné par* `LIB-filtre(`C`)` *est soit LIB-consistant, soit inconsistant sur les réels.*

En effet, l'algorithme retourne un ensemble de contraintes consistant sur les réels quand l'ensemble Q des variables à filtrer est vide. Par conséquent, lors de la dernière série d'itérations, aucun domaine n'a été réduit et donc, pour chaque variable $x_i \in X_{int}$, on a $u_i = \lfloor u_r \rfloor$ et $l_i = \lceil l_r \rceil$, ce qui implique que les bornes u_i et l_i de x_i soient des entiers qui appartiennent à l'espace des solutions sur les réels.

5 Implémentation des solveurs linéaires

La LIB-propagation a été mise en œuvre par une approche coopérative. Le programme est composé d'un moniteur, qui est chargé de la gestion de la file des variables à filtrer, et pour chaque variable entière $x_i \in X_{int}$, de deux solveurs $min[x_i]$ et $max[x_i]$ respectivement chargés de calculer les bornes minimale et maximale de x_i. Chaque solveur $min/max[x_i]$ est implémenté par l'algorithme du Simplexe.

5.1 *Un Simplexe incrémental*

A l'initialisation, chaque solveur $min/max[x_i]$ reçoit l'ensemble des contraintes initiales et calcule la valeur optimale correspondante que x_i peut prendre sur les réels. Si cette valeur n'est pas entière, il envoie à tous les autres solveurs (via le moniteur) un message contenant la nouvelle contrainte à ajouter. Ainsi, chaque solveur $min/max[x_i]$ reçoit de nombreuses nouvelles bornes calculées par d'autres solveurs min/max et doit mettre à jour en conséquence la valeur optimale de x_i. Si l'on ajoute simplement une nouvelle contrainte dans le Simplexe à chaque nouvelle borne reçue, la taille du Simplexe va rapidement augmenter, et les performances chuter. Aussi est-il nécessaire d'utiliser *un Simplexe avec bornes explicites* afin de prendre en compte directement ces nouvelles bornes [Chv83, Hol94, Bac95]. D'autre part, afin de minimiser le nombre de pivotages, on effectue une *optimisation sur le Simplexe dual* [Chv83]. Cette optimisation duale permet de repartir du dernier optimum calculé, généralement très proche du nouvel optimum à atteindre. Ainsi, *chaque nouvelle borne est prise en compte de façon incrémentale.*

La fonction LIB-filtre est exécutée à chaque étape de la recherche pour réduire des domaines. A chaque retour en arrière dans l'arbre de recherche, l'état des différents solveurs doit être restauré. Une première solution consiste à mémoriser à chaque point de choix l'état complet de chacun des solveurs. Cette solution est coûteuse en espace mémoire. La solution retenue dans l'implémentation actuelle consiste à mémoriser uniquement la valeur des bornes de chaque variable à chaque point de choix, et à recalculer incrémentalement à chaque retour en arrière l'état de chacun des solveurs par une optimisation duale du Simplexe.

5.2 *Instabilité numérique du Simplexe*

L'algorithme du Simplexe est complet sur les réels et les rationnels, mais pas sur les nombres flottants, du fait des erreurs d'arrondi. Afin de minimiser l'importance de ces erreurs d'arrondi, nous avons utilisé l'algorithme du *Simplexe révisé*, plus stable numériquement [Chv83, RH96]. Néanmoins, certaines erreurs d'arrondi peuvent survenir. De façon pragmatique, on ajoute (resp. soustrait) une valeur $\epsilon = 1.10^{-6}$ à chaque borne supérieure (resp. inférieure) calculée par un solveur linéaire. Cette solution permet d'éviter en pratique la majorité des erreurs d'arrondi. Cependant, d'un point de vue théorique, la seule solution totalement fiable serait d'utiliser les nombres rationnels.

Un prototype de l'algorithme de LIB-filtrage a été implémenté en C. On a réutilisé pour l'implémentation des solveurs linéaires certains modules du système `lp_solve` [Ber96]. Il est à noter que ce solveur, bien que basé sur l'algorithme du Simplexe révisé, n'est que moyennement fiable et échoue en moyenne sur la moitié des problèmes du fait d'instabilités numériques. En particulier, lorsque des erreurs numériques sont détectées alors que la matrice des coefficients vient juste d'être ré-inversée, le solveur `lp_solve` échoue.

6 Application de la LIB-consistance

On évalue dans cette section l'efficacité du LIB-filtrage pour résoudre des PLNEs par une recherche par séparation et évaluation. On compare

1. les résultats obtenus lorsque la phase de relaxation est effectuée par un solveur (appelé `LIB-solveur`), qui vérifie la LIB-consistance du système de contraintes, avec
2. les résultats obtenus lorsque la phase de relaxation est effectuée par un simple solveur linéaire sur les réels (appelé `LP-solveur`), qui vérifie la consistance de la relaxation continue du problème.

6.1 *Algorithme de recherche par séparation et évaluation*

On compare ces deux solveurs pour la résolution de PLNEs. On utilise dans les deux cas l'algorithme de recherche par séparation et évaluation décrit dans la figure 3.

fonction `sépare-et-évalue`(X_{int}, C, z)
retourne la solution optimale du PLNE (X_{int}, C, z)
  $Solution \leftarrow \infty$
  $Q \leftarrow \{(X_{int}, C, z)\}$
  tant que $Q \neq \emptyset$ **faire**
    enlever un PLNE (X_{int}, C', z) de Q
    si `relaxe`($C' \cup \{z < Solution\}$)=vrai **alors**
      soit $\{v_1, \ldots, v_n\}$ la solution minimisant z sous les contraintes C'
      si $\forall x_i \in X_{int}$, v_i est un nombre entier
      alors */* on a trouvé une solution entière */*
        $Solution \leftarrow$ valeur de z correspondant à $\{v_1, \ldots, v_n\}$
      sinon */* création de 2 nouveaux sous-PLNEs */*
        choisir une variable $x_i \in X_{int}$ telle que v_i ne soit pas entier
        $C'_{inf} \leftarrow C' \cup \{x_i \leq \lfloor v_i \rfloor\}$
        $C'_{sup} \leftarrow C' \cup \{x_i \geq \lceil v_i \rceil\}$
        $Q \leftarrow Q \cup \{(X_{int}, C'_{inf}, z), (X_{int}, C'_{sup}, z)\}$
      fin si
    fin si
  fin tant que
  retourner($Solution$)
fin `sépare-et-évalue`

Figure 3 : *Algorithme d'optimisation par séparation et évaluation*

Dans cet algorithme, la variable Q contient l'ensemble des PLNEs à résoudre. Au départ, Q contient l'unique PLNE initial. A chaque itération, un PLNE est enlevé de Q. Si sa relaxation est consistante et si sa solution optimale ne respecte pas toutes les contraintes d'intégralité, alors ce PLNE est remplacé dans Q par deux sous-PLNEs, où le domaine d'une variable a été séparé en deux. Dans l'implémentation actuelle, l'ensemble Q est implémenté par une pile (last-in / first-out) ce qui correspond à une exploration en profondeur d'abord de l'arbre de recherche.

La fonction `relaxe` est utilisée à chaque itération pour vérifier la consistance partielle du PLNE sélectionné. Cette fonction dépend du solveur choisi :

- quand `LP-solveur` est utilisé, la fonction `relaxe`(C) rend vrai si la relaxation continue de C est consistante (le solveur linéaire de `LP-solveur` est implémenté avec le même algorithme du Simplexe que les solveurs linéaires de `LIB-solveur`, c'est à dire un Simplexe révisé dual avec bornes explicites) ;

– quand `LIB-solveur` est utilisé, la fonction `relaxe`(C) filtre l'ensemble de contraintes C par la fonction `LIB-filtre`(C) décrite dans la figure 2, et rend vrai si la relaxation continue du système de contraintes résultant est consistante.

Les deux solveurs sont comparés en utilisant successivement les deux heuristiques suivantes concernant le choix de la variable x_i sur laquelle effectuer le branchement :

– l'heuristique `max-fract` sélectionne à chaque point de choix la variable x_i telle que v_i est la valeur ayant la partie fractionnelle la plus proche de 0.5, autrement dit, x_i est la variable "la moins entière" ;

– l'heuristique `min-dom` sélectionne à chaque point de choix la variable x_i ayant le plus petit domaine, autrement dit, x_i est la variable "la plus critique".

6.2 *Description du jeu d'essai*

`LIB-solveur` et `LP-solveur` sont comparés pour la résolution de PLNEs extraits de la bibliothèque MIPLIB [BBI92]. Cette bibliothèque, disponible électroniquement, répertorie une série de 59 PLNEs significatifs, souvent issus d'applications réelles, et permet de comparer les performances des différents codes d'optimisation.

Certains de ces problèmes, comportant généralement plus de 200 variables entières, n'ont pas pu être résolus en utilisant le LIB-filtrage, par manque d'espace mémoire. Pour pouvoir résoudre ces plus gros problèmes, on envisage d'implémenter le LIB-filtrage sur une architecture distribuée où les solveurs linéaires sont répartis sur plusieurs machines. Cet aspect est discuté dans le paragraphe 7.3. `LIB-solveur` et `LP-solveur` ont été comparés sur tous les autres problèmes de MIPLIB. Sur 13 de ces problèmes (à savoir : danoint, dcmulti, dsbmip, enigma, khb05250, lseu, misc03, misc07, modglob, noswot, pk1, qiu et stein45), la résolution a échoué du fait d'erreurs numériques successives (cf paragraphe 5.2). On peut noter que ces erreurs surviennent en moyenne aussi souvent lorsqu'on utilise `LP-solveur` que lorsqu'on utilise `LIB-solveur`, et pour beaucoup de ces problèmes dans les deux solveurs à la fois.

Finalement, l'ensemble des problèmes considéré est successivement décrit dans la figure 4 par le nombre de variables à valeurs sur les réels (R), le nombre

Problème	variables R	E	B	contraintes	optimum avec X_{int}	sur les réels
bell3a	62	32	39	123	878430.32	862578.64
bell5	36	28	30	91	8966406.49	8608417.95
egout	94	0	55	98	568.1	149.58
flugpl	6	11	0	17	1201500	1167185.75
gt2	0	164	24	29	21 166	13 460.23
p0033	0	0	33	14	3089	2520.60
p0033-10	0	19	14	14	2455	1734.17
pp08a	176	0	64	136	7350	2748.34
pp08aCUTS	176	0	64	246	7350	5480.60
rgn	80	0	100	24	82.19	48.79
stein27	0	0	27	118	18	13

Figure 4 : *Description des problèmes*

de variables à valeurs sur les entiers (E), le nombre de variables binaires à valeurs dans {0,1} (B), le nombre de contraintes du problème, la valeur optimale du problème en respectant les contraintes d'intégralité sur les variables entières et binaires, puis la valeur optimale en relaxant les contraintes d'intégralité.

Certains de ces problèmes (egout, p0033, pp08a, pp08aCUTS, rgn et stein-27) ne contiennent que des variables entières binaires. Par conséquent, ces problèmes n'ont pas été résolus avec l'heuristique `min-dom` (dans la mesure où tous les domaines des variables entières contiennent exactement 2 valeurs). Le problème p0033-10 est le même problème que p0033, mais où les domaines des variables ont été élargis à [0..10].

6.3 *Analyse des résultats*

Les deux solveurs sont successivement comparés dans la figure 5 sur le nombre de points de choix effectués, le nombre total d'itérations du Simplexe et le temps cpu. Pour chacun de ces critères, on donne les résultats obtenus en utilisant l'heuristique `max-fract` puis l'heuristique `min-dom`. Pour chaque heuristique, la première colonne donne les résultats obtenus avec `LIB-solveur`, la deuxième les résultats obtenus avec `LP-solveur` et la troisième le rapport entre `LP-solveur` et `LIB-solveur`.

Sur ces exemples, on remarque que le LIB-filtrage renforce effectivement la relaxation continue du problème et permet de réduire considérablement l'espace de recherche : en moyenne, le nombre de branches explorées est 120 fois plus faible avec `LIB-solveur` qu'avec `LP-solveur`. Néanmoins, ces résultats ne sont pas uniformes : dans certains cas (tels que bell3a avec l'heuristique `max-fract` ou flugpl avec l'heuristique `min-dom`), la LIB-propagation permet de réduire de plus de 200 fois le nombre de points de choix ; dans la majorité des cas, elle permet d'effectuer environ 100 fois moins de points de choix ; enfin, sur les 2 problèmes rgn et stein27, l'effet de la LIB-propagation est beaucoup moins important et ne réduit plus que 5 fois le nombre de branches à explorer.

Cependant, si la LIB-propagation réduit effectivement l'espace de recherche, elle est aussi plus coûteuse à mettre en œuvre. De fait, le contrôle d'un point de choix par `LIB-solveur` fait appel à plusieurs optimisations duales du Simplexe (2 par variable entière), alors que `LP-solveur` n'en nécessite qu'une seule. Chaque optimisation duale demande elle-même plusieurs itérations de l'algorithme du Simplexe. Ainsi, à chaque point de choix, `LIB-solveur` effectue plus de 100 itérations en moyenne tandis que `LP-solveur` n'en effectue que 4. Par conséquent, l'efficacité globale de la LIB-propagation dépend de son degré de réduction du nombre de points de choix : pour les problèmes où le LIB-filtrage réduit de plus de 100 fois le nombre de points de choix, `LIB-solveur` effectue de 3 à 15 fois moins d'itérations que `LP-solveur` ; pour les problèmes intermédiaires où le LIB-filtrage réduit de l'ordre de 80 fois l'espace de recherche, le nombre d'itérations effectuées devient comparable ; enfin, pour les 2 problèmes rgn et stein27, pour lesquels le LIB-filtrage ne divise que par 5 le nombre de points de choix, les rapports s'inversent et `LIB-solveur` affectue 5 fois plus d'itérations que `LP-solveur`.

Le temps cpu est essentiellement lié au nombre d'itérations du Simplexe. Ainsi, pour les problèmes rgn et stein27, la LIB-propagation, plus coûteuse qu'efficace, est 5 fois plus lente qu'un simple solveur linéaire. En revanche, pour tous les autres problèmes, elle permet d'accélérer de façon significative la résolution : de 1.6 fois pour egout avec l'heuristique `max-fract` jusqu'à plus de 6 fois pour bell5 et flugpl avec l'heuristique `min-dom`. En moyenne, sur l'ensemble des problèmes considérés, `LIB-solveur` est 3 fois plus rapide que `LP-solveur`.

Nombre de points de choix :

Problème	Heuristique `max-fract` LIB	LP	$\frac{LP}{LIB}$	Heuristique `min-dom` LIB	LP	$\frac{LP}{LIB}$
bell3a	4 407	892 414	202.5	2 991	122 667	41.0
bell5	> 30mn cpu			11 788	1 487 699	126.2
egout	384	33 090	86.2	-	-	-
flugpl	212	17 513	82.6	31	15 003	484.0
p0033	28	4 276	152.7	-	-	-
p0033-10	509	52 641	103.4	593	28 344	47.8
rgn	460	2 450	5.3	-	-	-
stein27	520	2 604	5.0	-	-	-

Nombre d'itérations du Simplexe :

Problème	Heuristique `max-fract` LIB	LP	$\frac{LP}{LIB}$	Heuristique `min-dom` LIB	LP	$\frac{LP}{LIB}$
bell3a	437 781	4 530 351	10.3	144 373	1 215 710	8.4
bell5	> 30mn cpu			755 969	11 429 974	15.1
egout	67 923	76 393	1.1	-	-	-
flugpl	36 859	40 922	1.1	6 344	28 789	4.5
p0033	6 225	20 956	3.4	-	-	-
p0033-10	89 143	248 644	2.8	53 755	126 252	2.4
rgn	516 617	91 979	0.2	-	-	-
stein27	230 994	30 806	0.1	-	-	-

Temps cpu (en secondes) :

Problème	Heuristique `max-fract` LIB	LP	$\frac{LP}{LIB}$	Heuristique `min-dom` LIB	LP	$\frac{LP}{LIB}$
bell3a	1 192.9	6 165.3	5.2	409.3	1 498.7	3.7
bell5	> 30mn cpu			1 692.4	10 892.4	6.4
egout	100.8	158.4	1.6	-	-	-
flugpl	10.4	18.2	1.8	2.4	15.2	6.3
p0033	2.4	7.1	3.0	-	-	-
p0033-10	33.5	83.0	2.5	21.7	42.3	2.0
rgn	245.6	42.1	0.2	-	-	-
stein27	417.9	73.7	0.2	-	-	-

Figure 5 : *Comparaison de* `LP-solveur` *et* `LIB-solveur`

Heuristique `max-fract` **:**

Problème	Sol. opt.	LIB	LP	$\frac{\text{LP}-\text{Sol.opt.}}{\text{LIB}-\text{Sol.opt.}}$
bell5	8 966 406	8 969 078	9 377 729	153.9
pp08a	7 350	8 290	9 760	2.6
pp08aCUTS	7 350	7 930	9 130	3.1
gt2	21 166	101 266	103 520	1.0

Heuristique `min-dom` **:**

Problème	Sol. opt.	LIB	LP	$\frac{\text{LP}-\text{Sol.opt.}}{\text{LIB}-\text{Sol.opt.}}$
gt2	21 166	106 378	135 468	1.3

Figure 6 : *Comparaison de* `LP-solveur` *et* `LIB-solveur`

La figure 6 donne les résultats obtenus pour les problèmes plus difficiles, non résolus après plus de 30 minutes de temps cpu. Pour ces problèmes, l'exécution a été interrompue au bout de 30 minutes, et la figure 6 indique successivement pour chaque heuristique : la solution optimale du problème, la meilleure solution trouvée avec `LIB-solveur`, la meilleure solution trouvée avec `LP-solveur` et le rapport entre ces deux solutions (en termes d'écart avec la solution optimale du problème).

Pour le problème gt2, les solutions trouvées avec `LIB-solveur` et `LP-solveur` sont comparables (et globalement très éloignées de la solution optimale). Pour les autres problèmes, les solutions trouvées avec la LIB-propagation sont très nettement plus proches de la solution optimale que celles trouvées avec `LP-solveur` (de 2.6 à 150 fois plus proches).

Enfin, l'heuristique `min-dom` donne, sur tous les problèmes sauf gt2, de meilleurs résultats que l'heuristique `max-fract`, avec `LIB-solveur` comme avec `LP-solveur`. Néanmoins, l'amélioration est légèrement plus sensible pour `LIB-solveur` que pour `LP-solveur`. Ce résultat peut s'expliquer par le fait que `LIB-solveur` filtre les domaines et réduit donc leur taille au cours de la résolution.

7 Discussion

7.1 *Résolution de problèmes mixtes sur les réels*

On définit dans [RS97] un solveur linéaire basé sur une architecture similaire à celle utilisée pour implémenter la LIB-propagation. Ce solveur utilise conjointement et de façon active un ensemble de solveurs linéaires sur les réels pour borner des variables. Ce solveur linéaire "actif" est intégré avec un solveur par filtrage de domaines dans un solveur de contraintes coopératif concurrent. Ce solveur, appelé `CCC`, permet la résolution de systèmes de contraintes mixtes sur les réels, comprenant des équations et inéquations linéaires et des équations non linéaires. Dans ce système, le solveur par filtrage de domaines réduit les domaines des variables par propagation d'intervalles sur les contraintes non-linéaires, tandis que le solveur linéaire calcule pour chaque variable son domaine de valeurs effectif par rapport aux contraintes linéaires. Les deux solveurs travaillent de façon concurrente et s'envoient mutuellement chaque nouvelle borne calculée.

7.2 *Une approche Branch & Cut*

L'approche Branch & Cut est souvent utilisée pour résoudre des problèmes linéaires sur les entiers. L'idée est d'ajouter itérativement de nouvelles contraintes linéaires qui "coupent" un certain nombre de solutions sur les réels, mais qui n'éliminent aucune solution sur les entiers. Le point clé dans cette approche est de trouver de "bonnes" coupes, c'est à dire des coupes qui se rapprochent le plus possible de l'enveloppe convexe du polyèdre des solutions entières. Dans le cas général, ce problème s'avère bien souvent aussi complexe que la résolution du système de contraintes. En revanche, en limitant la forme des contraintes, cette approche peut devenir très performante (e.g., CLP($\mathcal{PB}$) sur les pseudo-booléens [BB95]).

L'algorithme de filtrage que l'on propose ici est une forme (limitée) de Branch & Cut : les coupes engendrées par la LIB-consistance sont des bornes sur les variables. Ces coupes ne dépendent pas de la forme des contraintes et peuvent être déduites relativement simplement, sous réserve d'utiliser un algorithme du Simplexe incrémental. Ces coupes ont l'avantage de ne pas augmenter la taille du problème (i.e., le nombre de contraintes), sous réserve d'utiliser un algorithme du Simplexe avec bornes explicites.

Les solveurs coopératifs de [BB94, BB96] ainsi que les techniques locales de pré-traitement [Sav94] génèrent également des coupes portant sur les bornes des variables. Ces coupes sont déduites par une analyse locale des contraintes. Elles sont plus faciles à calculer, mais en contrepartie elles sont bien souvent moins profondes.

7.3 *Parallélisme "et"*

Quand on utilise un solveur sur une relaxation du problème à chaque point de choix de la recherche, un point clé réside dans le choix de la complétude du solveur : il s'agit de trouver un bon compromis entre un solveur coûteux qui réduit fortement l'espace de recherche et un solveur moins coûteux qui coupe moins de branches. Ainsi, si la LIB-propagation permet de couper beaucoup de branches, elle est également coûteuse, en temps et en espace mémoire. Les premières expérimentations montrent néanmoins qu'elle permet de résoudre des problèmes linéaires trois fois plus rapidement en moyenne qu'un simple appel à un solveur linéaire sur les réels.

En revanche, sur les plus gros problèmes, faisant intervenir plus de 200 variables entières ou binaires, la LIB-propagation demande beaucoup d'espace mémoire (2 Simplexes par variable entière) et ne peut être mise en œuvre efficacement sur une simple machine.

Une solution réside dans le fait que la LIB-propagation est très naturellement parallélisable : chaque solveur linéaire $min/max[x_i]$ peut s'exécuter indépendamment des autres, les seules communications entre solveurs concernant les nouvelles valeurs de bornes. Aussi envisage-t-on d'implémenter la LIB-propagation sur une architecture distribuée, permettant la résolution de PLNEs plus importants. Cette solution permettrait également d'accélérer la résolution globale d'un problème en accélérant la phase de LIB-filtrage effectuée à chaque noeud de la recherche. Ce parallélisme "ET" pourrait être comparé avec le parallélisme "OU", consistant à explorer plusieurs branches de l'arbre en parallèle [MR92].

Remerciements à Bruno De Backer pour l'envoi des problèmes de multi sac à dos, et à Jean-Marc Fouet, Simone Pimont et Michel Rueher pour la relecture des versions successives de ce rapport.

Bibliographie

[Bac95] B. De Backer. *Méthode de résolution de disjonctions de contraintes linéaires — application à la programmation logique avec contraintes —*. PhD thesis, Université d'Orléans, 1995.

[BB94] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. In C. Beierle and L. Plume, editors, *Logic Programming: formal methods and practical applications*. Elsevier Science Publisher, 1994.

[BB95] P. Barth and A. Bockmayr. Finite domain and cutting plane techniques in CLP($\mathcal{PB}$). In Leon Sterling, editor, *Proceedings of ICLP'95*, pages 133–147, Kanagawa, Japan, 1995. MIT Press.

[BB96] H. Beringer and B. De Backer. Cooperative solvers and global constraints: the case of linear arithmetic constraints. In *ILPS Workshop*, 1996.

[BBI92] R.E. Bixby, E.A. Boyd, and R.R. Indovina. MIPLIB: a test set of real-world mixed integer programming problems. *SIAM News*, 25(2):16, 1992. (available at ftp://softlib.cs.rice.edu/pub/miplib).

[Ben95] F. Benhamou. Interval Constraint Logic Programming. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*, pages 1–21. Springer-Verlag, 1995.

[Ber96] M. Berkelaar. lp_solve. Technical report, available at `ftp://ftp.es.ele.tue.nl/pub/lp_solve`, 1996.

[BMH94] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) Revisited. In *Proceedings of ILPS'94*, pages 124–138, Ithaca, NY, USA, 1994.

[Chv83] V. Chvatal. *Linear Programming*. W.H. Freeman and Co, 1983.

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fith Generation Computer Systems*, 1988.

[Dia95] D. Diaz. *Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: le système clp(FD)*. PhD thesis, INRIA, 1995.

[Hen95] P. Van Hentenryck. Constraint solving for combinatorial search problems. In *PPCP'95: Conference on Principles and Practice of Constraint Programming*. MIT Press, 1995.

[Hol94] C. Holzbaur. A specialized, incremental solved form algorithm for systems of linear inequalities. Technical Report TR-94-07, University of Vienna, 1994.

[HSD95] P. Van Hentenryck., V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). In *PPCP'95: Conference on Principles and Practice of Constraint Programming*. MIT Press, 1995.

[JNS97] E. Johnson, G. Nemhauser, and M. Savelsbergh. Progress in integer programming: an exposition. Research report, Georgia Institute of Technology, 1997.

[Lho93] O. Lhomme. Consistency techniques for numeric CSPs. In R. Bajcsy, editor, *Proceedings of the 13th IJCAI*, pages 232–238, Chambéry, France, 1993. IEEE Computer Society Press.

[MR92] B. Mans and C. Roucairol. Performances des algorithmes branch and bound parallèles à stratégie meilleur d'abord. Research Report RR-1716, INRIA, 1992.

[Pro96] PrologIA. PrologIV — Constraint inside (handbook of PrologIV). Technical report, PrologIA, Parc Technologique de Luminy, Case 919, 13288 Marseille Cedex 09, France, 1996.

[RH96] P. Refalo and P. Van Hentenryck. CLP($\mathcal{R}_{lin}$) revised. In *ICLP'96: Joint International Conference on Logic Programming*. MIT Press, 1996.

[RS97] M. Rueher and C. Solnon. Concurrent cooperating solvers over reals. *Journal of Reliable Computing, Kluwer academic publishers*, 3(3), 1997.

[Sav94] M.W.P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[SN93] M. Savelsbergh and G. Nemhauser. Functional description of MINTO, a Mixed INteger Optimizer. Technical Report COC-91-03A, Georgia Institute of Technology, 1993.

[Wal72] D.L. Waltz. Generating semantic descriptions from drawing of scenes with shadows. Technical Report AI-TR-271, MIT Cambridge, 1972.