Optimisation multicritère pour la gestion de dépendances logicielles : utilisation de la norme de Tchebycheff

Daniel Le Berre, Emmanuel Lonca, Pierre Marquis, Anne Parrain

Optimisation multicritère pour la gestion de dépendances logicielles : utilisation de la norme de Tchebycheff

Daniel Le Berre Emmanuel Lonca Pierre Marquis Anne Parrain

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

Résumé

Le problème de gestion de dépendances logicielles concerne l'installation d'applications informatiques modulaires. Il s'agit de programmes ayant la faculté d'être configurés par l'utilisateur, qui peut choisir à tout moment les modules qu'il souhaite installer ou enlever. Un module peut nécessiter la présence d'autres modules pour fonctionner, il peut entrer en conflit avec certains modules, et parfois il peut recommander l'installation de modules spécifiques pour pouvoir être utilisé au meilleur de ses capacités. Il existe généralement plusieurs solutions (listes de paquets à installer et à enlever) pour passer d'une configuration courante à une configuration souhaitée. On peut se donner des critères pour ordonner ces solutions, passant alors d'un problème de décision (« est-ce que je peux ajouter tous ces modules ? ») à un problème d'optimisation (« quelle est la meilleure installation permettant l'ajout de tous ces modules ? »). En outre, l'évaluation d'une installation est multicritère. Cet article se concentre sur des méthodes algorithmiques capables de calculer des solutions équilibrées en utilisant la norme de Tchebycheff comme méthode d'agrégation de critères. Cette approche est évaluée sur des problèmes de gestion de dépendances entre paquets GNU/Linux.

Mots Clef

Gestion de dépendances logicielles, norme de Tchebycheff, optimisation multicritère.

Abstract

The dependency management problem deals with the installation of modular application programs. Those pieces of software can be configured by the user, who can choose packages to be installed or removed at any time. A package may require other packages to work, may not be installable when other packages are installed, and might recommend some other packages to be installed to be fully featured. This problem generally admits several solutions (lists of packages to be installed or removed) to move from a current configuration to an expected configuration. The issue is to compute a preferred solution, so that we go from a decision problem ("can I install those packages ?") to an optimization one ("which solution is the best one to add all those packages ?"). In many cases, the user preferences are based on several criteria. This paper focuses on algorithmic methods to compute balanced solutions using Chebyshev distance as an aggregation method. This approach is assessed on GNU/Linux software dependency management.

Keywords

Software dependency management, Chebyshev distance, multicriteria optimization.

1 Introduction

Cet article traite d'un problème concret, la gestion de dépendances logicielles. Ce problème concerne l'installation de programmes informatiques modulaires. Un module peut nécessiter la présence d'autres modules pour pouvoir fonctionner, il peut entrer en conflit avec certains modules, et parfois il peut recommander l'installation de modules spécifiques pour pouvoir être utilisé au meilleur de ses capacités. Le problème est donc de déterminer, lorsqu'on veut installer ou mettre à jour un programme informatique, les modules à installer et les modules à enlever. Le problème de décision associé est NP-complet [8, 12]. Plusieurs approches pour résoudre ce problème ont été proposées ces dernières années dans le cadre des dépendances GNU/Linux [2].

A titre d'exemple, la figure 1 présente une instance simple du problème de gestion de dépendances dans le format CUDF défini par le projet Mancoosi [13]. L'utilisateur souhaite installer une machine web. Il existe deux configurations possibles : une version avec navigateur web et lecteur de courriels et une seconde version qui comporte aussi un gestionnaire de calendrier. Deux versions incompatibles de navigateurs web et de lecteurs de courriels sont disponibles. Ces dépendances peuvent facilement s'exprimer en logique propositionnelle. Par exemple, l'incompatibilité entre logiciels se traduit par une clause binaire négative ($\neg firefox_2 \vee \neg firefox_3$) et les dépendances par des clauses de Horn ($\neg webmachine_1 \vee web$,

```
PACKAGE : FIREFOX
VERSION : 2
PROVIDES : WEB
CONFLICTS : FIREFOX=3

PACKAGE : FIREFOX
VERSION : 3
PROVIDES : WEB
CONFLICTS : FIREFOX=2
INSTALLED : TRUE

PACKAGE : THUNDERBIRD
VERSION : 2
PROVIDES : MAIL
CONFLICTS : THUNDERBIRD=3

PACKAGE : THUNDERBIRD
VERSION : 3
PROVIDES : MAIL
CONFLICTS : THUNDERBIRD=2

PACKAGE : LIGHTNING
VERSION : 1
PROVIDES : CALENDAR

PACKAGE : WEB-MACHINE
VERSION : 1
DEPENDS : WEB, MAIL

PACKAGE : WEB-MACHINE
VERSION : 2
DEPENDS : WEB,MAIL,CALENDAR

REQUEST : INSTALLE UNE MACHINE WEB
INSTALL : WEB-MACHINE
```

FIGURE 1 – Exemple de problème d'installation au format CUDF

$\neg webmachine_1 \vee mail$). Des clauses non-Horn sont utilisées pour exprimer les choix entre logiciels ($\neg web \vee firefox_2 \vee firefox_3$).
$\{webmachine_1,\ firefox_2,\ thunderbird_3\}$ et $\{webmachine_2,\ firefox_3,\ thunderbird_3,\ lightning_1\}$ sont deux installations (ensembles de logiciels) satisfaisant ces dépendances. La première pourrait être préférée par un utilisateur souhaitant minimiser le nombre de logiciels à installer alors que la seconde pourrait être préférée par un utilisateur souhaitant toujours disposer de la dernière version d'un logiciel. Dans cet exemple, $firefox_2$ est déjà installé sur le système, donc la première installation minimise les changements de logiciels.

Dans le cas où l'on considère un unique critère pouvant être exprimé par une fonction objectif linéaire, il existe des programmes (« solveurs ») capables de calculer une solution préférée en un temps raisonnable. Malheureusement, il est souvent impossible de réaliser une modélisation efficace des attentes d'un décideur en ne se basant que sur un unique critère. Par exemple, dans le cas du problème de gestion de dépendances, on peut vouloir installer un minimum de paquets (pour ne pas utiliser trop d'espace disque, par exemple), mais il est aussi souhaitable que les paquets installés soient le plus à jour possible (afin de profiter des dernières fonctionnalités apportées par ce programme). On entre alors dans la problématique de la décision multicritère. Dans un tel cadre, une solution peut être très satisfaisante pour un critère et très peu satisfaisante pour un autre et nous voulons éviter de telles solutions.

Pour définir et calculer des solutions « équilibrées », nous nous sommes intéressés à la norme de Tchebycheff. Nous avons implanté et expérimenté des algorithmes d'optimisation existants pour calculer des solutions préférées au sens de Tchebycheff dans le cadre de notre problème. Ceux-ci se révélant inadaptés, nous avons développé et évalué de nouveaux algorithmes pour calculer des solutions équilibrées pour les problèmes de gestion de dépendances.

2 Choisir une « bonne solution »

Le projet européen Mancoosi a mis en avant une approche pour résoudre les problèmes de gestion de dépendances logicielles basée sur une technique d'optimisation avec un ordre lexicographique des critères, chaque critère étant modélisé par une fonction objectif linéaire et l'ensemble des solutions réalisables étant caractérisé par un ensemble C de contraintes pseudo-booléennes [5, 1]. L'optimisation avec ordre lexicographique sur les critères est une méthode d'agrégation de critères simple à mettre en œuvre et dont le temps d'exécution est raisonnable. L'idée consiste à trier les différents critères selon un ordre d'importance (strict et total) et d'éliminer, via l'adjonction de contraintes, pour chaque critère pris par ordre d'importance décroissant, les solutions sous-optimales. Ainsi, sont gardées successivement les solutions optimales pour le premier critère, puis celles pour le deuxième critère parmi celles qui sont optimales pour le premier, et ainsi de suite, comme illustré à la figure 2. L'intérêt de ce type de méthode est que les fonctions objectifs à optimiser sont linéaires puisqu'à chaque itération de l'algorithme, un seul critère est considéré ; on obtient ainsi des temps d'exécution raisonnables. De plus, les solutions obtenues sont nécessairement Pareto-optimales (il n'existe pas de solution strictement meilleure).

Cependant, cette méthode fournit généralement des solutions déséquilibrées ; ainsi, les critères les moins prioritaires peuvent être sacrifiés pour permettre d'améliorer, même faiblement, un critère plus prioritaire. Ceci est un défaut important dans notre cadre. Nous nous sommes donc intéressés à des méthodes permettant d'obtenir des solutions plus balancées.

Supposons que nous souhaitions installer un programme, et qu'il existe plusieurs solutions au problème de gestion de dépendances associé. Celles-ci sont comparées à l'aide de deux critères à minimiser : le nombre de paquets installés (c_1), et le nombre de paquets changés, *i.e.* non installés devenus installés ou vice-versa (c_2). Voici trois solutions

caractérisées par leur valeur sur (c_1, c_2) :

$$\begin{aligned} s_a &= (33, 43) \\ s_b &= (36, 40) \\ s_c &= (35, 41) \end{aligned}$$

On remarque que les solutions s_a et s_b sont chacune la meilleure sur un des critères, mais la pire sur l'autre. En revanche, la solution s_c est plus balancée. En effet, elle n'est la meilleure pour aucun des critères, mais elle limite la perte sur les deux critères. Plutôt que de nous intéresser aux valeurs des critères, nous considérons les différences entre les valeurs de ces critères et les meilleures valeurs possibles pour chacun des critères. On peut maintenant juger le fait qu'une solution soit équilibrée ou non en calculant sa distance d_i au point idéal $(33, 40)$ pour chaque critère c_i :

$solution$	(c_1, c_2)	d_1	d_2	$\max(d_1, d_2)$
s_a	$(33, 43)$	0	3	3
s_b	$(36, 40)$	3	0	3
s_c	$(35, 41)$	2	1	2

Il existe diverses manières d'agréger ces distances. L'intuition de base est qu'une solution équilibrée est relativement bonne pour tous les critères, c'est-à-dire que sa distance au point idéal est faible pour chacun des critères. Ainsi nous pouvons retenir les solutions dont la plus grande de ces distances est plus faible que celle des autres solutions. La plus grande des distances pour chaque solution est donnée par $max(d_1, d_2)$. Dans notre exemple, c'est la solution s_c qui apparaît comme la plus équilibrée.

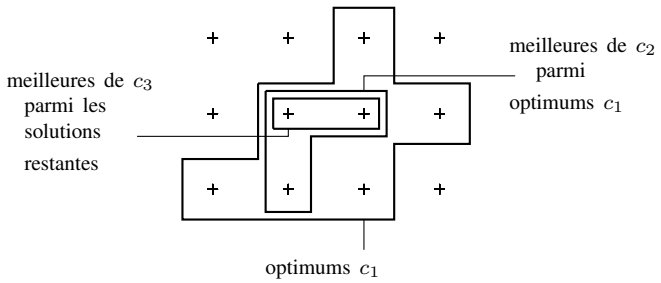

FIGURE 2 – Optimisation lexicographique sur 3 critères (c_1, c_2, c_3) (dans l'ordre)

Cette définition de la distance entre une solution et le point idéal comme étant la plus grande des distances à chacun des critères correspond à la distance de Tchebycheff.

3 Norme de Tchebycheff

La norme de Tchebycheff permet de caractériser des solutions (ou alternatives réalisables) équilibrées.

Définition 1 (alternative réalisable)**.** *Soit $A = A_1 \times A_2 \times ... \times A_m$ l'ensemble des alternatives ($A = \mathbb{B}^m$ dans notre cadre). Une alternative réalisable est une alternative qui satisfait les contraintes du problème.*

Les alternatives réalisables sont évaluées en fonction de critères, on peut donc associer à une alternative un vecteur de valeurs (dans $\mathbb{N}dansnotrecadre$) pour ces critères. Un même vecteur de valeurs peut correspondre à plusieurs alternatives. La valeur d'une solution pour un critère est donnée par l'image de cette solution par une fonction objectif qui exprime un coût. On compare deux alternatives réalisables en comparant leurs vecteurs de valeurs. On appelle **point idéal** le vecteur des valeurs optimales pour chacune des fonctions de coût parmi les alternatives réalisables. Celui-ci ne correspond bien souvent pas à une alternative réalisable. La méthode de Tchebycheff se base sur la norme de Tchebycheff entre le vecteur de valeurs de l'alternative courante et ce point idéal.

Définition 2 (point idéal)**.** *Soit S l'ensemble des alternatives réalisables, et soit $c_1, c_2, ..., c_n$ les fonctions de coût associées aux n critères. Le point idéal $(c_1^*, c_2^*, ..., c_n^*) \in \mathbb{N}^n$ est le vecteur tel que :*

$$\forall i \in \{1, 2, ..., n\},\ c_i^* = min(\{c_i(s) \mid s \in S\}).$$

Pour un critère donné, la différence entre la valeur pour une solution et celle du point idéal exprime le **regret** de choisir cette solution, pour le critère.

Définition 3 (regret)**.** *Soit $x^* = (c_1^*, c_2^*, ..., c_n^*)$ le point idéal d'un problème à n critères et $c_1, c_2, ..., c_n$ les fonctions de coût associées à ces critères. Soit $s \in S$ une solution du problème. Le regret de s pour le $i^{ème}$ critère est l'entier : :*

$$r_i(s) = c_i(s) - c_i^*$$

Nous pouvons maintenant définir la méthode **min-max regret**, qui préfère dans l'ensemble S les solutions qui offrent le plus petit regret maximal sur les critères considérés.

Définition 4 (min-max regret)**.** *Soient s et s' deux alternatives réalisables, et $r_1, r_2, ..., r_n$ les fonctions exprimant le regret sur les n critères. La solution s est préférée à la solution s' par la méthode du **min-max regret** si et seulement si :*

$$\begin{gathered} max(\{r_i(s) \mid i \in 1, ..., n\}) \\ < \\ max(\{r_i(s') \mid i \in 1, ..., n\}) \end{gathered}$$

Une variante de cette méthode où les regrets associés à chaque critère sont pondérés est nommée **méthode de Tchebycheff** [11], [14].

Définition 5 (norme et méthode de Tchebycheff)**.** *Soient s et s' deux alternatives réalisables, et $r_1, r_2, ..., r_n$ les fonctions exprimant le regret sur les n critères. Soit $W = (w_1, w_2, ..., w_n)$ des poids (dans $\mathbb{N}$) associés aux n critères. La **norme de Tchebycheff** de s, aussi appelée max-regret de s, est donnée par :*

$$U(s) = max(\{w_i r_i(s) \mid i \in 1, ..., n\})$$

*La solution s est préférée à la solution s' par la **méthode de Tchebycheff** si et seulement si :*

$$U(s) < U(s')$$

La figure 3 illustre les notions introduites dans le cas où deux critères sont considérés. c_1 et c_2 sont des fonctions de coût, x^* (en rouge) est le point idéal du problème, tous les autres points sont des vecteurs de valeurs de solutions réalisables. Les solutions préférées selon un ordre lexicographique $c_1 > c_2$ sur les critères (en bleu) sont les solutions Pareto-optimales (leurs vecteurs de valeurs constituent la zone grise) dont les vecteurs de valeurs se trouvent strictement au-dessus ou à droite du point idéal. Les solutions préférées selon la norme de Tchebycheff (leurs vecteurs de valeurs sont en vert) sont celles dont les vecteurs de valeurs se trouvent sur le périmètre d'un carré le plus petit possible dont le coin inférieur droit est le point idéal. On peut noter que les solutions Tchebycheff-préférées ne sont pas forcément Pareto-optimales.

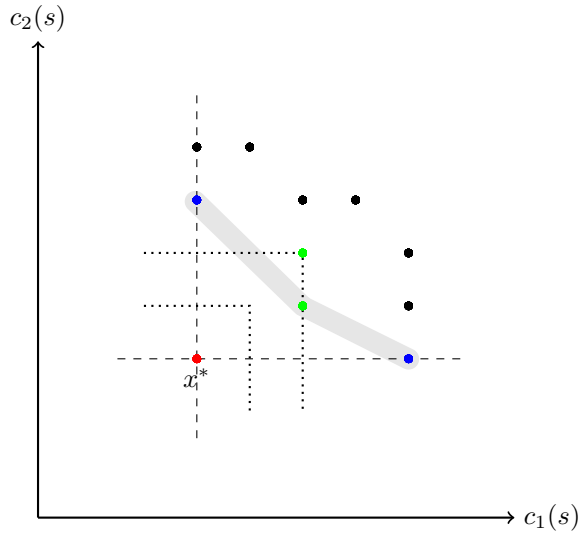

FIGURE 3 – Point idéal et vecteurs de valeurs de solutions préférées

4 Calcul de solutions optimales

4.1 Optimisation guidée par fonction minorante

L'objectif est de déterminer une alternative s^* qui satisfait toutes les contraintes de C et qui minimise U, la difficulté étant que U n'est pas une fonction linéaire.
Une solution algorithmique à ce problème a été proposée par Gonzales *et al.* [4] qui présentent une approche par fonction minorante. Soit un ensemble d'alternatives réalisables $S = \{s_0, s_1, ..., s_n\}$. Soit U_m une fonction linéaire telle que $U_m(s) \leq U(s)$ pour toute solution $s \in S$. L'idée est d'énumérer l'ensemble des solutions $\{s_0, s_1, ..., s_n\}$ selon la valeur de leur image par la fonction U_m, de la plus petite valeur à la plus grande. Il s'agit d'un problème d'optimisation linéaire pour lequel des algorithmes relativement efficaces existent ; le temps de calcul reste en général contenu dans le cas où on ne doit pas énumérer une grande quantité de solutions sous-optimales.
Il y a deux avantages à guider la recherche de solution optimale pour U comme expliqué ci-dessus :
– si la fonction U_m est une bonne approximation de la fonction U, alors les solutions optimales de U se trouvent parmi les solutions ayant une valeur proche de la valeur optimale de U_m, nous permettant de renvoyer une valeur sans doute proche de la valeur optimale de U en cas d'arrêt provoqué de l'algorithme (interruption due à un temps d'exécution trop long) ;
– il n'est pas nécessaire de parcourir l'ensemble des solutions admissibles dans la plupart des cas : on peut déterminer qu'une solution optimale a été trouvée sans énumérer toutes les solutions. En effet un cas d'arrêt est atteint lorsque la valeur de U_m pour s_i est supérieure ou égale à la meilleure valeur de U trouvée jusqu'ici [4].

Une difficulté de cette approche est qu'elle nécessite de disposer d'une bonne fonction minorante, c'est-à-dire une fonction linéaire qui est assez proche de la fonction objectif visée. Nous avons réalisé des expérimentations avec la moyenne arithmétique des regrets pondérés comme fonction minorante (la moyenne des regrets est bien inférieure ou égale à leur maximum). Malheureusement, cette fonction ne semble pas adaptée à notre cas d'étude. Sur tous les tests que nous avons effectués, nous avons observé que le comportement de la moyenne est celui d'une fonction par paliers : de nombreuses solutions du problème ont une même valeur moyenne pour les critères. L'algorithme doit donc parcourir toutes les solutions pour ces paliers, sans se rapprocher de la fonction objectif. Nous supposons que c'est le fait de se situer dans un cadre booléen qui induit ce comportement (peu de valeurs distinctes pour la moyenne). Des pistes restent à explorer pour déterminer une fonction minorante linéaire plus adaptée.

4.2 Ajout de contraintes bornant la valeur de la norme de Tchebycheff

Le problème de décision associé à la gestion de dépendances étant NP-complet, il est possible d'utiliser un codage en logique propositionnelle de ce problème [7, 2]. Dans ce cas, les modèles des contraintes C sont les solutions du problème et le problème traité est un problème d'optimisation sous contraintes pseudo-booléennes.
La définition de la norme de Tchebycheff permet de caractériser simplement la valeur $U(s^*)$ car majorer le maximum d'un ensemble revient à majorer tous les éléments du dit ensemble. Formellement, soit une valeur r telle que

$$C \cup \{w_i r_i(s) \leq r \mid i \in 1, \ldots, n\}$$

est satisfaisable, alors que

$$C \cup \{w_i r_i(s) < r \mid i \in 1, \ldots, n\}$$

ne l'est pas. Alors on a nécessairement $r = U(s^*)$.
Cela nous donne deux possibilités pour l'optimisation :
– optimiser par *renforcement* de contraintes : on cherche des solutions toujours meilleures en décrémentant r,

la valeur courante du max-regret autorisé ; c'est une approche classique pour les moteurs d'optimisation booléens (PB, MAXSAT) [9] ;
– optimiser par *relaxation* de contraintes : on cherche une solution avec un regret nul, puis on relâche cette contrainte en incrémentant le seuil r du max-regret autorisé tant qu'une solution n'est pas trouvée ; cette approche rencontre un vif succès dans le cadre du problème d'optimisation MAXSAT lorsqu'elle est couplée à la détection de noyau incohérent [10].

Optimisation par renforcement de contraintes. L'idée de cet algorithme itératif est de partir d'une solution quelconque, et de restreindre l'espace de recherche tant que c'est possible. On exploite ici le fait que toutes les fonctions de coût considérées ainsi que leurs poids sont à valeurs dans $\mathbb{N}$ (ce qui nous assure que la valeur de la norme de Tchebycheff est elle-aussi à valeurs dans $\mathbb{N}$). Ainsi, au départ, nous recherchons simplement une solution au problème, indépendamment des valeurs qu'elle donne aux différents critères. Soit r la valeur de son max-regret. On renforce alors les contraintes du problème en fixant $r-1$ comme majorant pour chacun des $w_i r_i(s)$. Si une solution existe, alors on itère le processus en utilisant le max-regret de cette nouvelle solution comme nouvelle valeur pour r. Dans le cas contraire, on conclut qu'il n'existe pas de solution s telle que $U(s) \in [0, r-1]$, et par conséquent les solutions qui donnent un max-regret de r sont optimales. L'algorithme (algorithme 1) est illustré à la figure 4 : lorsqu'une solution est déterminée, les contraintes ajoutées sont représentées sous forme d'une zone grise montrant quelles vecteurs de valeurs deviennent inaccessibles.

Entrées : problème P
Sorties : solution réalisable s minimisant U
$s \leftarrow obtenirSolution(U)$;
tant que $s \neq UNSAT$ **faire**
  $r \leftarrow maxRegret(P, s)$;
  $meilleure \leftarrow s$;
  $imposerMaxRegret(P, r)$;
  $s \leftarrow obtenirSolution(P)$;
fin
retourner *meilleure* ;

Algorithme 1: Renforcement de contraintes

Pour que cet algorithme soit correct, il faut que chaque valeur prise par la fonction objectif admette un prédécesseur, et que chacune de ces valeurs admette un nombre fini de minorants, ce qui est le cas quand les fonctions de coût c_i sont à valeur dans $\mathbb{N}$.
Le temps de calcul de cet algorithme est fortement dépendant de la valeur $U(s)$ du modèle s déterminé à la première itération. En effet, si cette valeur est n, l'algorithme effectuera au maximum n itérations. Une amélioration possible consiste à rechercher une solution relativement proche de la valeur optimale dès le début. Pour cela, nous avons

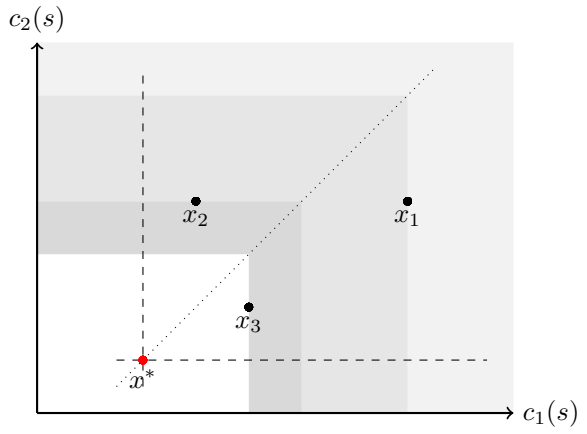

FIGURE 4 – Optimisation par renforcement de contraintes

« fusionné » l'approche à base de fonction minorante et celle à base de renforcement de contraintes. En effet, la fonction minorante étant en quelque sorte une approximation de la fonction objectif, on peut espérer réduire significativement le temps de calcul d'une solution optimale pour U par ajout de contraintes en choisissant comme solution initiale une bonne solution pour notre fonction d'approximation. On peut en plus de cela donner un temps limite d'exécution au solveur d'optimisation monocritère, afin d'éviter de perdre trop de temps lorsque celui-ci se rapproche de la valeur optimale pour la fonction minorante.

Optimisation par relaxation de contraintes. Il s'agit de l'approche duale de la précédente : on considère au départ une valeur de r qui minore $U(s^*)$. Si

$$C \cup \{w_i r_i(s) \leq r \mid i \in 1, \ldots, n\}$$

possède une solution, alors cette solution est optimale. Dans le cas restant, on « relâche » les contraintes imposées en remplaçant

$$\{w_i r_i(s) \leq r \mid i \in 1, \ldots, n\}$$

par

$$\{w_i r_i(s) \leq r+1 \mid i \in 1, \ldots, n\}$$

de manière à tester la valeur de r immédiatement supérieure à la précédente. On itère ce processus. Lorsqu'on trouve une solution, elle est nécessairement optimale (sinon on aurait trouvé une solution lors d'une itération passée). Voir l'algorithme 2 et la figure 5. Dans cette figure, les zones grises représentent les vecteurs de valeurs rendus successivement accessibles par la relaxation des contraintes.
Pour que cet algorithme soit correct, il faut disposer d'une valeur qui minore $U(s^*)$ (c'est le point de départ et dans notre cadre la valeur 0 convient), et faire varier r de façon à garantir que $U(s^*)$ sera atteint après un nombre fini d'itérations. Cela est le cas quand les fonctions de coût c_i sont à valeur dans $\mathbb{N}$ et que l'incrément choisi pour r est 1.

Entrées : problème P
Sorties : solution réalisable s minorant U
$s \leftarrow UNSAT$;
$r \leftarrow 0$;
répéter
  $meilleure \leftarrow s$;
  $imposerMaxRegret(P, r)$;
  $s \leftarrow obtenirSolution(P)$;
  $retirerRegretsImposés(P)$;
  $r \leftarrow r + 1$;
jusqu'à $s \neq UNSAT$;
retourner *meilleure* ;

Algorithme 2: Relaxation de contraintes

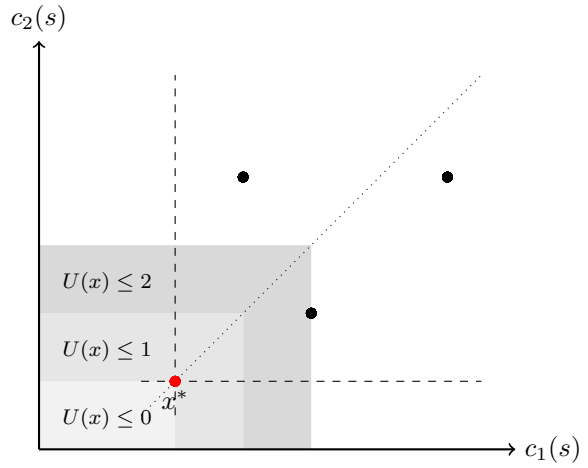

FIGURE 5 – Optimisation par relaxation de contraintes

5 Expérimentations et résultats

Nous avons implémenté en Java les différentes approches présentées dans une version modifiée de p2cudf [2] (une adaptation du logiciel p2 [7] capable de résoudre des problème de gestion de dépendances pour GNU/Linux), lui-même utilisant sat4j [6] (une librairie de solveurs pour des problèmes de décision ou d'optimisation). Plus précisément, les solveurs testés sont p2cudf ainsi que deux de ses variantes. La première utilise l'algorithme de relaxation de contraintes, alors que la deuxième implémente l'algorithme de renforcement de contraintes, avec un temps de recherche de dix secondes dédié au calcul d'une solution initiale. Nous ne présentons pas de résultats pour l'approche basée sur la fonction minorante : celle-ci n'a jamais été capable de terminer dans le temps imparti (elle est donc toujours battue par nos approches). Nos expérimentations ont été réalisées sur un Quad-core Intel XEON X5550 avec 32Go de mémoire. Nous avons testé nos algorithmes sur des instances issues des compétitions *misc* organisées dans le cadre du projet mancoosi [1]. Nous nous sommes intéressés à deux séries d'instances : les *dudf-real* de la compétition *misc2011*, et les *9orless* de la compétition *misc-live*. Il s'agit de problèmes réels d'installation de paquets sous la distribution GNU/Linux Debian. La série *9orless* contient des instances qui sont difficiles pour les solveurs. Les résultats sont présentés dans les tableaux 1, 2 et 3. Nous avons comparé la qualité des solutions obtenues (en utilisant le regret maximal sur les critères comme échelle) ainsi que le temps d'exécution des différents solveurs, qui est donné en *minutes, secondes*.
La légende du tableau est la suivante :

opt. / optimums : optimums pour chacun des critères pris séparément ;

p2cudf : p2cudf original (ordre lexicographique) ;

MR : max-regret ;

relaxation : p2cudf modifié implémentant l'algorithme de relaxation de contraintes ;

renforcement : p2cudf modifié implémentant l'algorithme de renforcement de contraintes ;

p2cudf-inv : version originale de p2cudf où l'ordre sur les critères a été inversé.

Les tests sont réalisés en tenant compte de deux critères : les solutions doivent minimiser le nombre de paquets enlevés, et minimiser le nombre de paquets changés (paquets non installés devenus installés ou vice-versa). Les échelles de ces critères étant identiques, nous n'avons pas à faire face à des problèmes de commensurabilité (les poids w_1 et w_2 valent 1). Pour l'optimisation lexicographique de p2cudf, le critère « removed » a été considéré comme plus prioritaire que le critère « changed » (agrégation nommée « paranoid » dans le projet mancoosi). Pour les solveurs basés sur le renforcement et la relaxation de contraintes, le calcul des optimums monocritères (nécessaires au calcul des regrets c_i^*) sont réalisés par des solveurs d'optimisation lancés en parallèle. Après ces calculs préliminaires, le reste du programme est séquentiel.
En ce qui concerne les instances *dudf-real* (tableau 1), l'ensemble des solveurs fournit le plus souvent des solutions de qualité identique. En effet, pour la plupart de ses instances, le point idéal correspond à une alternative réalisable et celle-ci constitue la réponse fournie par tous les solveurs. Bien qu'on ne tire pas avantage à utiliser une méthode multicritère sur de telles instances, on remarque qu'on ne perd pas significativement de temps à utiliser un solveur basé sur une telle méthode (on s'intéresse ici au temps réel, pas au temps CPU cumulé). On remarque aussi que sur le peu d'instances où le point idéal ne correspond pas à une solution réalisable, nos solveurs retournent comme attendu des solutions plus équilibrées.
Pour les instances *9orless*, le constat est différent : les solutions fournies par nos solveurs sont systématiquement « plus équilibrées » que celles données par p2cudf (tableau 2). Ce gain est avéré sur la plupart des instances et il peut être très significatif. En revanche, le temps de calcul devient beaucoup plus grand que pour la version originale de p2cudf. Il faut cependant noter que le temps de calcul de p2cudf dépend de l'ordre des critères considéré. En inversant les deux critères utilisés, on obtient des temps de

calcul comparables aux méthodes « balancées », i.e., renforcement et relaxation (tableau 3).

6 Conclusions et perspectives

Dans cet article, nous nous sommes attachés au problème de la modélisation et du calcul de solutions équilibrées pour le problème de gestion de dépendances logicielles. Après avoir défini une notion de solution équilibrée dans le cadre multicritère, nous nous sommes tournés vers la méthode de Tchebycheff pour l'agrégation de critères. Nous avons testé une méthode d'optimisation générale pour ce type de fonction d'agrégation non linéaire. Nous n'avons pas obtenu de bons résultats avec la fonction minorante appliquée habituellement, à savoir la moyenne des regrets : l'algorithme devait itérer sur un trop grand nombre de solutions avant de pouvoir retourner une solution optimale.
Nous avons donc proposé deux nouvelles méthodes spécifiques à notre problème codé dans le formalisme booléen. Ces méthodes, les optimisations par *renforcement* et *relaxation* de contraintes, inspirées d'algorithmes existants pour résoudre des problèmes d'optimisation booléenne, nous ont permis d'obtenir des solutions équilibrées, en un temps de calcul raisonnable.
Si nos premiers résultats sont encourageants, de nombreuses perspectives de travail sont ouvertes. D'une part, les méthodes proposées ne retournent pas nécessairement des solutions Pareto-optimales (il peut exister des solutions strictement meilleures). Nous espérons améliorer nos résultats en utilisant une méthode pour obtenir des solutions Pareto-optimales qui raffinent celle de Tchebycheff : le min leximax regret, dont le principe est de choisir le vecteur dont les coûts ordonnés sont les plus faibles. Des techniques de calcul pour le leximin/leximax ont été présentées par Bouveret *et al.* [3]. Cependant, elles ne sont pas utilisables directement dans le cadre booléen car le codage proposé s'appuie sur des variables à valeurs entières. Ce problème peut être pallié par des techniques de codage des variables entières en base 2, et nous envisageons donc d'adapter de telles méthodes à notre cadre et de les évaluer. D'autre part, actuellement, lorsqu'on multiplie le nombre de critères, le temps de calcul explose sur nos instances. Nous devons en étudier les raisons et vérifier si le problème vient des critères employés ou de nos algorithmes, et tenter d'améliorer cette situation.

Remerciements

Nous remercions les relecteurs anonymes pour leurs remarques constructives, qui nous ont permis d'améliorer la présentation de nos résultats. Nous remercions également le projet Mancoosi de mettre à disposition de la communauté des benchmarks.

Références

[1] `http://www.mancoosi.org/misc/`.

[2] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In Inês Lynce and Ralf Treinen, editors, *LoCoCo*, volume 29 of *EPTCS*, pages 11–22, 2010.

[3] S. Bouveret and M. Lemaître. Computing leximin-optimal solutions in constraint networks. *Artificial Intelligence*, 173(2) :343–364, 2009.

[4] Christophe Gonzales, Patrice Perny, and Sergio Queiroz. Preference aggregation with graphical utility models. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1037–1042. AAAI Press, 2008.

[5] G. Guttierez, M. Janota, I. Lynce, O. Lhomme, V. Manquinho, J. Marques-Silva, and C. Michel. Final version of the optimizations algorithms and tools. Technical report, Mancoosi (Managing the Complexity of the Open Source Infrastructure), 2011.

[6] D. Le Berre and A. Parrain. The sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010.

[7] D. Le Berre and P. Rapicault. Dependency management for the eclipse ecosystem : eclipse p2, metadata and resolution. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 21–30. ACM, 2009.

[8] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE*, pages 199–208. IEEE Computer Society, 2006.

[9] Olivier Roussel and Vaso Manquinho. Pseudo-boolean and cardinality constraints. In *Handbook of Satisfiability*, pages 695–733. IOS Press, February 2009.

[10] João P. Marques Silva. Minimal unsatisfiability : Models, algorithms and applications (invited paper). In *ISMVL*, pages 9–14. IEEE Computer Society, 2010.

[11] R.E. Steuer and E.U. Choo. An interactive weighted tchebycheff procedure for multiple objective programming. *Mathematical Programming*, 26(3) :326–344, 1983.

[12] Tommi Syrjänen. A rule-based formal model for software configuration. Research Report A55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999.

[13] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (cudf) 2.0. Technical report, Mancoosi, 2009.

[14] A.P. Wierzbicki. On the completeness and constructiveness of parametric characterizations to vector optimization problems. *OR Spectrum*, 8(2) :73–87, 1986.

instance	opt.	p2cudf	MR	relaxation	renforcement	MR
103c9978-...	0,0	0/0	0	0,0	0,0	0
19890cfe-...	0,0	0/0	0	0,0	0,0	0
1aabfc32-...	0,9	0/13	4	2,9	2,9	2
26f3d4cc-...	0,2	0/2	0	0,2	0,2	0
27000e82-...	0,2	0/2	0	0,2	0,2	0
29180036-...	0,0	0/0	0	0,0	0,0	0
2c3aece6-...	0,0	0/0	0	0,0	0,0	0
33bb2fbc-...	0,8	0/8	0	0,8	0,8	0
3e4f8550-...	0,41	0/41	0	0,41	0,41	0
4a69cf16-...	0,2	0/2	0	0,2	0,2	0
4e539b28-...	0,2	0/2	0	0,2	0,2	0
4ede8d96-...	0,0	0/0	0	0,0	0,0	0
5698a62c-...	0,2	0/2	0	0,2	0,2	0
56ae4afa-...	0,83	0/83	0	0,83	0,83	0
56e31304-...	0,0	0/0	0	0,0	0,0	0
6b0d1da0-...	0,2	0/2	0	0,2	0,2	0
7bf50d1c-...	0,0	0/0	0	0,0	0,0	0
7c834c0e-...	4,46	4/52	6	7,49	7,49	3
- cumul (temps réel) -		3 :17.58		3 :39.11	3 :22.47	
- cumul (temps CPU) -		3 :17.58		5 :06.21	5 :14.36	

TABLE 1 – Expérimentations sur les instances *dudf-real* – 18 parmi 59 instances

instance	optimums	p2cudf	MR	p2cudf-inv	MR	relaxation	renforcement	MR
kjumpingcube	33,40	33/43	3	36/40	3	35,42	35,41	2
kmines	33,40	33/43	3	36/40	3	35,42	35,41	2
knetwalk	33,40	33/43	3	36/40	3	35,42	35,41	2
kommander	33,40	33/43	3	36/40	3	35,42	35,41	2
kphotoalbum	33,40	33/43	3	36/40	3	35,42	35,41	2
ksystemlog	33,41	33/44	3	36/41	3	35,43	35,43	2
ktouch	33,40	33/43	3	36/40	3	35,42	35,41	2
kwin-style-crystal	33,41	33/44	3	36/41	3	35,43	35,42	2
libevolution3.0-cil	33,51	33/120	69	38/51	5	36,54	36,54	3
libgnokii4	34,48	34/51	3	37/48	3	36,50	36,49	2
libmono-addins-gui0.2-cil	33,61	33/121	60	38/61	5	36,64	36,64	3

TABLE 2 – Expérimentations sur les instances *9orless* (qualité des solutions) – 11 parmi 38 instances

instance	p2cudf	p2cudf-inv	relaxation	renforcement
kjumpingcube	0 :53.62	2 :12.22	2 :55.16	2 :31.18
kmines	0 :58.16	2 :16.17	2 :52.77	2 :38.40
knetwalk	0 :50.77	2 :19.50	2 :56.83	2 :18.94
kommander	0 :51.26	2 :11.65	3 :00.37	2 :33.43
kphotoalbum	0 :53.40	3 :21.69	3 :16.28	2 :14.66
ksystemlog	0 :43.02	2 :04.78	2 :45.00	1 :53.71
ktouch	0 :52.27	2 :18.82	2 :51.26	2 :41.91
kwin-style-crystal	0 :41.24	1 :32.96	2 :09.44	1 :11.90
libevolution3.0-cil	0 :28.21	4 :41.94	4 :11.74	5 :29.42
libgnokii4	0 :39.56	2 :58.75	3 :21.75	2 :10.52
libmono-addins-gui0.2-cil	0 :31.88	6 :15.23	5 :22.29	8 :41.16
- cumul (temps réel) -	30 :53.56	129 :53.26	130 :05.02	130 :50.06
- cumul (temps CPU) -	30 :53.56	129 :53.26	143 :20.80	152 :33.70

TABLE 3 – Expérimentations sur les instances *9orless* (temps de calcul) – 11 parmi 38 instances