Résolution du problème WCSP par extraction de noyaux insatisfiables minimaux

Christophe Lecoutre **Nicolas Paris** **Olivier Roussel** **Sébastien Tabary**

CRIL - CNRS UMR 8188,
Université Lille Nord de France, Artois,
rue de l'universite,
62307 Lens cedex, France
{lecoutre,paris,roussel,tabary}@cril.fr

Résumé

Les méthodes usuelles de résolution pour le cadre WCSP utilisent des techniques de transferts de coûts couplées à une recherche arborescente de type séparation et évaluation. Dans cet article, nous nous intéressons à une approche intégrant l'extraction et le relâchement de noyaux insatisfiables minimaux afin de résoudre ce problème, soit de manière incomplète (gloutonne), soit de manière complète.

1 Introduction

Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) consiste à déterminer si un réseau de contraintes donné (CN pour Constraint Network), ou instance CSP, est satisfiable ou non. Il s'agit d'un problème de décision, où on recherche une instantiation complète des variables du CN satisfaisant toutes les contraintes de celui-ci. Lorsque des préférences doivent être représentées, il est possible d'exprimer celles-ci à l'aide de contraintes souples. Dans le cadre WCSP (Weighted CSP), chaque contrainte souple associe en fait un coût aux différentes instanciations des variables sur lesquelles elle porte, permettant ainsi de modéliser différents degrés de violation. L'objectif est de trouver une instanciation qui minimise le coût cumulé des différentes contraintes souples.

La plupart des méthodes actuelles de résolution de réseaux de contraintes pondérées (WCN pour Weighted Constraint Network), ou instances WCSP, se fondent sur une recherche par séparation et évaluation (Branch and Bound) et utilisent diverses cohérences locales pour estimer le coût minimum du réseau simplifié au cours de la recherche. Ces cohérences locales (AC* [10, 11], FDAC [3], EDAC [6], VAC [4], OSAC [5]) sont fondées sur des méthodes de transferts de coût qui préservent l'équivalence. Parce qu'ils doivent prendre en compte plus d'informations que dans le cadre CSP, les algorithmes établissant les cohérences locales souples sont souvent plus complexes que leurs équivalents CSP.

Dans cet article, nous proposons une approche originale pour le cadre WCSP, consistant pour un WCN donné à résoudre une séquence de CNs générés itérativement à partir de ce WCN, et permettant ainsi la réutilisation de solveurs de contraintes éprouvés. Ce type d'approche a déjà été utilisée avec succès pour établir la cohérence souple VAC [4] : l'établissement de la cohérence d'arc sur un CN généré à partir d'un WCN a pour objectif (contrairement à notre approche) l'identification de transferts de coûts.

Pour résoudre un WCN, nous durcissons les contraintes en ne retenant que les tuples ayant un coût donné afin d'obtenir des contraintes dures (CNs). Le principe de la méthode est alors d'énumérer ces CNs par ordre croissant de coût (au niveau du WCN). Lorsqu'un CN est insatisfiable, un noyau minimalement inconsistant (MUC pour Minimal Unsatisfiable Core) est identifié pour déterminer les contraintes souples pour lesquelles il faut accepter un coût plus important afin d'obtenir une solution. Cette approche est déclinée en un algorithme complet et un algorithme glouton (donc incomplet). Notons que cette approche a été utilisée avec succès pour le cadre SAT [1, 8, 13].

Cet article est construit comme suit. Après les définitions d'usage, nous présentons le fonctionnement gé-

néral des algorithmes et les structures de données utilisées. Nous détaillons alors la version complète ainsi que la version gloutonne de notre approche et terminons par la présentation de quelques résultats expérimentaux.

2 Préliminaires

2.1 Généralités

Un *réseau de contraintes* (CN) P est constitué d'un ensemble fini de n variables noté $vars(P)$ et d'un ensemble fini de e contraintes, noté $cons(P)$. Chaque variable x a un domaine associé noté $dom(x)$, qui contient l'ensemble fini des valeurs pouvant être assignées à x ; le domaine initial de x est noté $dom^{init}(x)$. Chaque contrainte dure c porte sur un ensemble ordonné de variables, noté $scp(c)$ et appelé *portée* de c. Elle est définie par une relation contenant l'ensemble des tuples autorisés pour les variables de $scp(c)$. L'arité d'une contrainte c est le cardinal de $scp(c)$. Une *instanciation* I d'un ensemble $X = \{x_1, \ldots, x_p\}$ de variables est un ensemble $\{(x_1, a_1), \ldots, (x_p, a_p)\}$ tel que $\forall i \in 1..p, a_i \in dom^{init}(x_i)$. I est *valide* sur P ssi $\forall (x, a) \in I, a \in dom(x)$. Une solution est une instanciation complète de $vars(P)$ (i.e., l'assignation d'une valeur à chaque variable) qui satisfait toutes les contraintes. P est satisfiable ssi il admet au moins une solution. Pour plus d'information sur les réseaux de contraintes, voir [7, 12, 15].

Un noyau insatisfiable de P est un sous-ensemble insatisfiable de contraintes de P. Un noyau est un MUC (Minimal Unsatisfiable Core) si et seulement si tout sous-ensemble strict du MUC est satisfiable. Une approche composée de deux étapes pour extraire les MUCs de réseaux de contraintes est présentée dans [9]. Montrée comme la plus efficace parmi celles présentées dans l'article précédemment cité, nous avons opté dans notre implantation pour la version *dichotomique*, nommée *dcMUC*.

Un *réseau de contraintes pondérées* (WCN) W est constitué d'un ensemble fini de n variables noté $vars(W)$, un ensemble fini de e contraintes souples, noté $cons(W)$, et d'une valeur k qui est soit un entier naturel strictement positif soit $+\infty$. Chaque contrainte souple $w \in cons(W)$ possède une portée $scp(w)$ et est définie par une fonction de coût de $l(scp(w))$ vers $\{0, \ldots, k\}$, où $l(scp(w))$ est le produit cartésien des domaines des variables sur lesquelles porte w ; pour toute instanciation $I \in l(scp(w))$, on notera le coût de I dans w par $w(I)$. Une instanciation de coût k (noté aussi $\top$) est interdite. Autrement, elle est autorisée avec le coût correspondant (0, noté aussi $\bot$, est complètement satisfaisant). Les coûts sont combinés par la somme bornée $\oplus$ définie par :

$$\forall a, b \in \{0, \ldots, k\}, a \oplus b = min(k, a + b)$$

L'objectif du problème de satisfaction de contraintes pondéré (WCSP) est, pour un WCN donné, de trouver une instanciation complète de coût minimal. Pour plus d'information sur les contraintes pondérées, voir [2, 14].

Différentes variantes de la cohérence d'arc souple pour le cadre WCSP ont étés proposées durant ces dix dernières années. Il s'agit de AC* [10, 11], la cohérence d'arc directionnelle complète (FDAC) [3], la cohérence d'arc directionnelle existentielle (EDAC) [6], la cohérence d'arc virtuelle (VAC) [4] et la cohérence d'arc souple optimale (OSAC) [5]. Tous les algorithmes proposés pour atteindre ces différents niveaux de cohérence utilisent des opérations de transfert de coûts (ou transformations préservant l'équivalence) telles que la projection unaire, la projection et l'extension.

2.2 Strates et fronts

Dans ce papier, nous considérons des contraintes souples données en extension. Il s'agit donc de contraintes tables souples, où certains tuples sont listés explicitement, accompagnés de leur coût, tandis qu'un coût par défaut indique le coût des tuples implicites (non représentés). Nous introduisons ci-dessous les notions de strate et front.

Les tuples d'une contrainte souple ayant le même coût peuvent être regroupés dans un même sous-ensemble S nommé *strate*. Le coût des tuples d'une strate S est donné par $cost(S)$. Une strate particulière correspond au coût par défaut : cette strate ne contient aucun tuple explicite, mais représente implicitement tous les tuples qui n'apparaissent pas explicitement dans une strate. Le nombre de strates d'une contrainte w sera désigné par $nbLevels(w)$. Au sein d'une contrainte, nous considérons les strates ordonnées par coût strictement croissant. Par souci de simplicité, nous utiliserons des indices (de 0 à $nbLevels(w) - 1$) pour identifier les strates d'une contrainte, et quand le contexte est suffisamment clair confondrons les indices avec les strates qu'ils représentent.

Un *front* est une fonction qui à chaque contrainte d'un WCN associe l'une de ses strates. Ce front représente une frontière entre des strates qui seront autorisées à un instant donné de notre approche, et celles qui seront interdites. Nous utiliserons une notation de tableau pour les fronts. Ainsi, $f[w]$ représentera la strate associée à la contrainte w dans un front f. $cost(f)$ est la fonction qui, à un front f, associe la somme des coûts des strates désignées par ce front :

$$cost(f) = \sum_{w \in cons(W)} cost(f[w]).$$

Un front f' est le successeur direct d'un front f s'il existe une contrainte w_j telle que $f'[w_j] = f[w_j] + 1$ et $\forall i \neq j, f'[w_i] = f[w_i]$. On choisit de noter $f \rightarrow f'$ si f' est un successeur direct de f et $f \rightarrow_* f'$ s'il existe une suite de relations $f \rightarrow f_1, f_1 \rightarrow f_2, \ldots f_n \rightarrow f'$. Comme pour une contrainte donnée, les strates ont des coûts strictement croissants, on en déduit que :

$$f \rightarrow_* f' \Rightarrow cost(f) < cost(f').$$

On notera que les fronts avec la relation successeur direct ont une structure de treillis, où le plus petit front (noté $\bot$) désigne pour toute contrainte sa strate de plus faible coût et le plus grand front (noté $\top$) désigne pour toute contrainte sa strate de plus grand coût.

3 Fonctionnement général

La figure 1 décrit le fonctionnement général de l'approche que nous proposons dans cet article pour la résolution de réseaux de contraintes pondérées. Pour commencer, un premier réseau de contraintes P est construit (via l'appel à la fonction `toCN`) à partir du réseau de contraintes pondérées W à résoudre : pour chaque contrainte de W, les tuples apparaissant dans la strate de coût minimum sont considérés comme autorisés par P tandis que tous les autres tuples sont considérés comme interdits. Cette méthode de construction est similaire à celle proposée dans [4]. Le réseau de contraintes P ainsi construit est alors résolu par un solveur de contraintes (via l'appel à la fonction `solveCN`). Si P n'est pas satisfiable, l'algorithme extrait un MUC (via l'appel à la fonction `extractMUC`) à partir de P.

Le MUC extrait de cette façon peut être utilisé soit dans une version gloutonne, soit dans une version complète.

Dans la version gloutonne, certaines contraintes du MUC vont être successivement relâchées jusqu'à restaurer la satisfiabilité (via l'appel à la fonction `relax`). Plus précisément un sous-ensemble des strates de certaines contraintes du MUC vont basculer du statut "interdit" au statut "autorisé". Lorsque la satisfiabilité du MUC est restaurée, les relaxations sont retranscrites dans le réseau de contraintes initial et le processus se répète jusqu'à établir la satisfiabilité globale du CN. Dans ce cas, une solution est retournée.

Dans la version complète, une fois le MUC identifié, on insère dans une file de priorité une représentation d'un ensemble de réseaux de contraintes à résoudre correspondant aux différentes manières de relâcher le MUC précédemment identifié. Les CNs seront ainsi résolus successivement jusqu'à l'identification d'un réseau de contraintes satisfiable. Dans ce cas, une solution optimale est retournée.

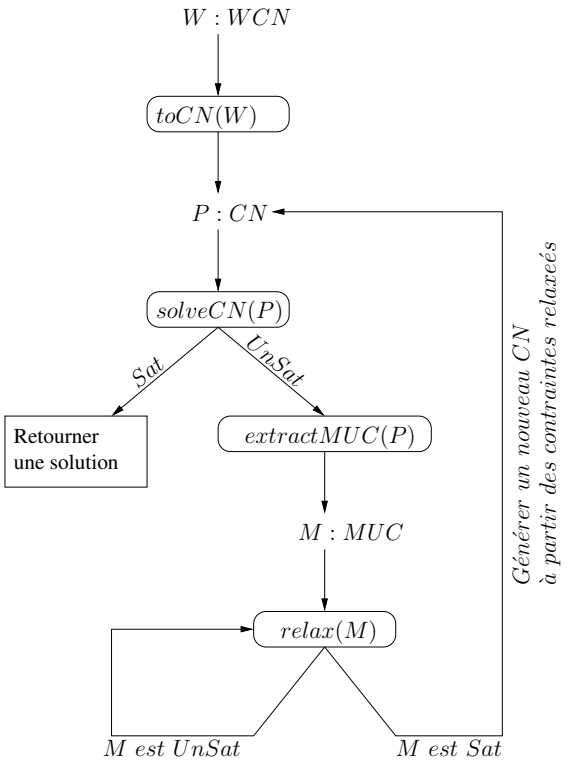

FIGURE 1 – Principe de fonctionnement du Relâchement itératif de MUC.

4 Structures de données et fonctions

Le tableau $front$ est utilisé par l'opération `toCN` qui construit un CN à partir d'un WCN et des strates sélectionnées. En fait, il existe deux versions de cette opération. La première version notée $\texttt{toCN}_{=}(W, front)$ construit un CN en ne retenant comme tuples autorisés pour une contrainte w du WCN W que les tuples de la strate $front[w]$; cette strate est dite autorisée. La seconde version notée $\texttt{toCN}_{\leq}(W, front)$ construit un CN en ne retenant comme tuples autorisés pour une contrainte w du WCN W que les tuples des strates d'indice inférieur ou égal à $front[w]$ (donc de coût inférieur ou égal) ; ces strates sont dites autorisées. Autrement dit, pour toute contrainte dure c construite à partir d'une contrainte souple w de W, c n'autorise que les tuples dont le coût est égal (respectivement, inférieur ou égal) au coût de la strate $front[w]$.

En pratique, il y a deux manières de gérer une contrainte dure. D'une part, quand les (ou la) strates autorisées ne contiennent pas la strate correspondant au coût par défaut, `toCN` se contente de créer une contrainte dure positive qui liste les tuples de ces strates autorisées comme étant des tuples autorisés. D'autre part, quand les (ou la) strates autorisées incluent la strate correspondant au coût par défaut, `toCN`

crée une contrainte dure négative qui interdit tous les tuples des strates non autorisées, et ceci afin d'éviter de devoir énumérer tous les tuples ayant le coût par défaut.

Nous considérons maintenant chaque contrainte souple w comme un objet que nous décrivons. Tout d'abord, $w.defaultCost$ désigne le coût par défaut associé aux tuples implicites de la contrainte. Ensuite, les strates de la contrainte w sont stockées dans le tableau $w.levels$. Pour une strate d'indice i donnée, on accède d'une part au coût des tuples par $w.levels[i].cost$ et d'autre part à la liste des tuples par $w.levels[i].tuples$. Ce tableau de strates est trié par ordre de coûts strictement croissants. Le nombre de strates pour w est obtenu simplement par l'expression $nbLevels(w)$. Pour finir, le tableau $front$, déjà évoqué précédemment, fait correspondre à chaque contrainte w l'indice de la strate $front[w]$ qui, selon l'approche employée, sera soit la seule autorisée, soit la plus grande strate autorisée. Ces structures sont notamment utilisées dans les fonctions `toCN` et `cost`.

La figure 2 décrit pour deux contraintes w_0 et w_1 les structures de données présentées ci-dessus. La contrainte w_0 possède un coût par défaut $w_0.defaultCost$. Le tableau $w_0.levels$ permet d'accéder aux différentes strates dans lesquelles sont stockés les tuples de même coût. La contrainte w_0 possède un ensemble de strates numérotées à partir de 0. Les strates étant triées par ordre de coût strictement croissant, la strate $w_0.levels[0]$ contient l'ensemble des tuples de coût minimum (dont le coût est accessible par $w_0.levels[0].cost$). La liste des tuples de la deuxième strate, à savoir (τ_2, τ_3), est accessible par $w_0.levels[1].tuples$. Dans cet exemple $front[w_0]$ est égal à 1, ce qui signifie que, en fonction de l'opération $toCN$ utilisée, 1) soit sont autorisées les strates dont l'indice est ≤ 1 (c'est à dire l'ensemble des tuples de la première et de la seconde strate soit τ_1, τ_5, τ_7, τ_2 et τ_3), 2) soit sont autorisées les strates dont l'indice est égal à 1 (c'est à dire l'ensemble des tuples de la seconde strate soit τ_2 et τ_3). Pour la contrainte w_1, nous avons fait apparaître le coût par défaut au niveau de la deuxième strate (indice 1). Rappelons que cette strate est ajoutée de manière à conserver la structure $levels$ triée par coût strictement croissant, afin de simplifier les traitements algorithmiques. Comme déjà indiqué, les tuples implicites associés à ce type de strates ne sont pas représentés, et ceci est symbolisé dans la figure par le symbole $\emptyset$.

Nous terminons cette section en décrivant les fonctions utilisées par nos algorithmes, mais non détaillées dans le papier. La fonction `solveCN` résout un CN donné en paramètre et retourne soit une solution, soit $\bot$ si le CN est insatisfiable. La fonction `extractMUC`

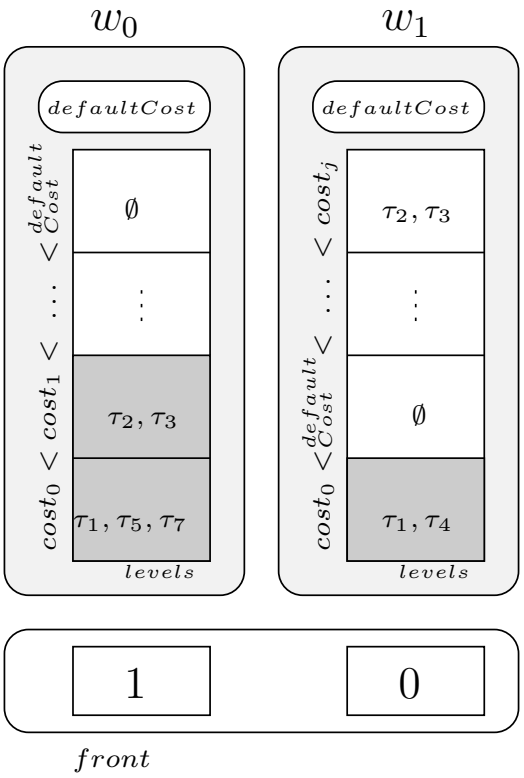

FIGURE 2 – Description des structures de données.

retourne un MUC (sous-ensemble de contraintes minimalement insatisfiable) à partir d'un CN P passé en paramètre (celui-ci doit être insatisfiable). La fonction `cons`(W, M) retourne les contraintes du WCN W qui correspondent à celles du MUC M (toute contrainte dure est associée à une contrainte souple au moment de la génération). La fonction `restrict`(W, M) retourne un WCN qui contient uniquement les contraintes du WCN W qui correspondent à des contraintes présentes dans le MUC M.

5 Algorithmes

5.1 Version complète préliminaire

Pour faciliter la compréhension de notre approche, nous présentons un premier algorithme complet qui n'exploite pas les MUC et n'est donc pas efficace. La fonction completeSearchNoMUC commence avec un premier front qui ne retient que la strate 0 de chaque contrainte (ligne 1 à 3). Ensuite, tant qu'il reste un front à étudier, on extrait d'une file de priorité celui de plus faible coût. Notons que le coût d'un front peut être facilement calculé en sommant les coûts associés aux strates identifiées au niveau de chaque contrainte souple. On peut alors construire avec l'opérateur `toCN`$_=$ un CN que l'on cherche à résoudre avec

`solveCN`. Si une solution est obtenue, elle est optimale, et l'algorithme peut s'arrêter. En effet, l'algorithme garantit une énumération des fronts par ordre de coût croissant (voir proposition 1) et assure que tous les fronts précédents étaient insatisfiables. Si le CN s'est avéré insatisfiable, on énumère les successeurs directs du front courant (ligne 10 à 15) et on les insère dans la file de priorité. Bien évidemment, il faut vérifier que l'on ne dépasse pas la strate maximale d'une contrainte donnée. Par ailleurs, les fronts qui auraient un coût égal à k doivent être ignorés.

Function completeSearchNoMUC(W : WCN)

```
1  foreach w ∈ cons(W) do
2      front[w] ← 0
3  Q ← {front}
4  while Q ≠ ∅ do
5      f ← pick and delete least cost front in Q
6      P ← toCN_=(W, f)
7      sol ← solveCN(P)
8      if sol ≠ ⊥ then
9          return
10     else
11         foreach w ∈ cons(W) do
12             if f[w] ≠ nbLevels(w) − 1 then
13                 f' ← f
14                 f'[w] ← f'[w] + 1
15                 if cost(f') ≠ k then
16                     Q ← Q ∪ f'
```

La file de priorité Q utilisée dans cet algorithme est une version particulière car elle doit non seulement garantir que les fronts sont extraits par ordre croissant de coût, mais également garantir qu'un front donné n'est pas inséré deux fois dans la file. Par exemple, si l'on considère un réseau de 2 contraintes ayant chacune au moins 2 strates, le premier front sera le tableau $< 0, 0 >$. Lors de la première itération, on insérera dans la file les voisins $< 1, 0 >$ et $< 0, 1 >$. Lorsque ces fronts seront à leur tour extraits de la file, ils généreront tous deux le même voisin $< 1, 1 >$ qui ne doit être inséré qu'une seule fois dans la file pour éviter des calculs redondants. Cette file particulière s'obtient en modifiant très légèrement une implantation classique d'une file de priorité par tas binomial.

On peut vérifier aisément que l'algorithme completeSearchNoMUC énumère tous les fronts possibles quand `solveCN` ne trouve jamais de solution. En effet, en faisant abstraction des coûts, il s'agit d'une exploration en largeur d'abord d'un arbre énumérant les fronts. La proposition 1 garantit en plus que cet algorithme ne peut pas boucler et que les fronts sont énumérés par ordre de coût croissant.

Proposition 1 *Les deux propriétés suivantes sont vérifiées par l'algorithme completeSearchNoMUC :*

A) les fronts sont énumérés par ordre croissant de coût ;

B) une fois qu'un front f est extrait de la file de priorité, il ne peut plus jamais y être inséré.

Preuve 1 Soit f_1 un premier front extrait de la file et f_2, un front extrait de la file après l'extraction de f_1.

A) Deux cas sont possibles. Soit (A.1) f_2 se trouvait déjà dans la file quand f_1 en a été extrait, soit (A.2) f_2 a été placé dans la file après l'extraction de f_1. Dans le cas (A.1), la file de priorité garantit que $cost(f_1) \leq cost(f_2)$. Dans le cas (A.2), f_2 est issu d'un front f tel que $f \rightarrow_* f_2$ et tel que soit $f = f_1$, soit f était présent dans la file quand f_1 a été extrait. Dans les deux cas, $cost(f_1) \leq cost(f)$. Comme $f \rightarrow_* f_2 \Rightarrow cost(f) < cost(f_2)$, on en conclut que $cost(f_1) < cost(f_2)$. On obtient donc dans tous les cas que $cost(f_1) \leq cost(f_2)$.

B) Supposons que $f_1 = f_2$. Comme la file garantit qu'elle ne contient jamais deux fronts identiques à un instant donné, on en conclut que l'on se situe nécessairement dans le cas (A.2) précédent. Or, on a prouvé que dans ce cas, $cost(f_1) < cost(f_2)$ ce qui contredit $f_1 = f_2$. □

Il est à noter que dans notre approche, les contraintes unaires sont considérées comme des contraintes tout à fait ordinaires. Par ailleurs, dans la mesure où une seule strate d'une contrainte est retenue à un instant donné, les CN générés sont de taille beaucoup plus réduite que le WCN de départ. On peut espérer que le test de satisfiabilité soit assez simple (donc rapide) à réaliser dans la plupart des cas. Une dernière observation est que cet algorithme exploite une forme d'abstraction en considérant que, du point de vue du coût global, il n'y a pas lieu de distinguer entre eux les tuples de même coût d'une même contrainte.

Pour finir, l'inconvénient de cette première version est qu'elle énumère toutes les combinaisons possibles de strates. Dans le pire des cas, elle énumère un nombre de combinaisons égal au produit du nombre de strates de chaque contrainte, soit $\prod_{w \in cons(W)} nbLevels(w)$. Selon le nombre de variables et de contraintes, cette complexité peut s'avérer nettement plus grande que dans une approche Branch and Bound classique (i.e. quand $\prod_{w \in cons(W)} nbLevels(w) \ggg \prod_i |dom(x_i)|$). Cela s'explique en particulier par le fait qu'une même instanciation peut être explorée plusieurs fois dans les différents tests de satisfiabilité. Néanmoins, même dans ce cas,

cette approche peut se révéler payante si le coût de l'optimum est faible.

5.2 Exploitation des MUC

La complexité de l'algorithme précédent peut être fortement réduite en pratique en remarquant qu'il est vain de passer à la strate supérieure pour une contrainte qui ne participe pas à l'insatisfiabilité du CN. L'idée est donc d'identifier un MUC et de ne passer à la strate supérieure que pour les contraintes faisant partie de ce MUC.

Dans le pire des cas, le MUC retourné peut contenir toutes les contraintes du problème et donc la complexité dans le pire des cas n'est pas changée. En pratique cependant, sur des instances concrètes, les MUC sont souvent de petite taille. De ce fait, le voisinage généré est bien plus petit et la complexité pratique est fortement réduite.

5.2.1 Approche complète

La fonction completeSearch (inspirée de [1, 8, 13]) présente l'algorithme modifié pour prendre en compte les MUC dans une version complète. Les premières étapes de l'algorithme sont les mêmes que pour completeSearchNoMUC. Quand le CN courant est insatisfiable un MUC est identifié, et on va progressivement autoriser des strates plus élevées dans les contraintes du MUC. Plus précisément, l'algorithme énumère toutes les manières possibles de relâcher ce MUC. Pour chaque contrainte du MUC (obtenue par l'appel $\texttt{cons}(W, M)$), l'algorithme génère un successeur f' du front courant qui ne diffère de ce dernier que par l'incrémentation de la strate autorisée de cette contrainte. Ce nouveau front est alors inséré dans la file Q à une position dépendante de la valeur de $cost(f')$.

Function completeSearch(W : WCN)

```
1  foreach w ∈ cons(W) do
2      front[w] ← 0
3  Q ← {front}
4  while Q ≠ ∅ do
5      f ← pick and delete least cost front in Q
6      P ← toCN_=(W, f)
7      sol ← solveCN(P)
8      if sol ≠ ⊥ then
9          return sol
10     else
11         M ← extractMUC(P)
12         foreach w ∈ cons(W, M) do
13             if f[w] ≠ nbLevels(w) − 1 then
14                 f' ← f
15                 f'[w] ← f'[w] + 1
16                 if cost(f') ≠ k then
17                     Q ← Q ∪ {f'}
```

Proposition 2 *L'algorithme completeSearch est complet.*

Preuve 2 Par abus de langage, on utilisera le terme front pour désigner implicitement le CN associé à ce front. Soit f_u le front qui est identifié comme insatisfiable et M le MUC extrait de ce front. La seule différence entre completeSearch et completeSearchNoMUC est que lorsqu'un MUC est identifié, on restreint la recherche en s'imposant de relâcher d'abord ce MUC.

Si le WCN n'admet pas de solution, restreindre ainsi la recherche ne fait pas perdre de solution. On peut donc se restreindre au cas où le WCN admet au moins une solution S. Soit f_S le front correspondant à une quelconque solution S. L'algorithme completeSearchNoMUC garantit qu'il existe toujours un front f dans la file de priorité tel que $f \rightarrow_* f_S$. C'est en particulier vrai pour le premier front placé dans la file $\bot$ et lorsque cette propriété est vérifiée pour un front extrait de la file, elle est vérifiée pour au moins l'un de ses successeurs direct (par définition de $\rightarrow_*$). Donc, par récurrence, cette propriété est toujours garantie.

Si $f_u \not\rightarrow_* f_s$, la restriction proposée ne peut pas nous faire perdre la solution S. On peut donc faire l'hypothèse supplémentaire que $f_u \rightarrow_* f_s$. De ce fait, $\forall w \in cons(W), f_s[w] \geq f_u[w]$. Par ailleurs, il existe nécessairement une contrainte $w_i \in cons(W, M)$ t.q. $f_s[w_i] > f_u[w_i]$, faute de quoi f_S serait toujours insatisfiable. Soit f' le front défini par $f'[w_i] = f_u[w_i] + 1$ et $\forall w \in cons(W)$ t.q. $w \neq w_i$, $f'[w] = f_u[w]$. f' est un front qui est généré par l'algorithme et par construction $f' \rightarrow_* f_s$. □

La figure 3 présente un exemple où on ne peut pas se contenter de relâcher un MUC d'une seule manière, fût-ce le relâchement de coût optimal. Dans cet exemple, un WCN est composé de trois contraintes w_x, w_{xy} et w_y. Le premier front sélectionne les strates de coût 0, ce qui ne conserve comme tuples autorisés que $\{(x, a)\}$, $\{(x, a), (y, b)\}$ et $\{(y, a)\}$. Bien évidemment, les contraintes w_{xy} et w_y forment dans ce cas un MUC. Il y a deux manières de le relâcher : la première consiste à passer à la strate suivante de w_{xy} (c'est localement le meilleur choix), et la seconde à la strate suivante de w_y. Dans le premier cas, le nouveau front obtenu conserve comme tuples autorisés $\{(x, a)\}$, $\{(x, c), (y, a)\}$ et $\{(y, a)\}$. Cette fois, le MUC porte sur

w_x et w_{xy}. On peut soit relâcher w_{xy} pour aboutir à une solution ayant un coût de 100, soit relâcher w_x pour aboutir à un autre MUC que l'on peut relâcher de deux manières pour aboutir soit à une solution de coût 105, ou à une solution de coût 110.

Si dans le premier MUC on avait relâché w_y, on aurait obtenu directement une solution de coût 10 qui est l'optimum.

coût	tuple
100	{(x,c)}
10	{(x,b)}
0	{(x,a)}

(a) w_x

coût	tuple
100	default
5	{(x,c),(y,a)}
0	{(x,a),(y,b)}

(b) w_{xy}

coût	tuple
100	{(y,c)}
10	{(y,b)}
0	{(y,a)}

(c) w_y

FIGURE 3 – De la nécessité d'énumérer tous les relâchements d'un MUC.

5.2.2 Approche gloutonne

Nous présentons maintenant une version incomplète de notre approche utilisant l'extraction et le relâchement de MUC pour la résolution de WCNs.

À partir d'un WCN W, la fonction incompleteSearch retourne une solution d'un CN dérivé de W. La structure $front$ est tout d'abord initialisée à 0, ce qui correspond à la première strate de chaque contrainte du WCN.

À partir du WCN W et de la structure $front$ on extrait un CN et ce réseau de contraintes est ensuite résolu. Si une solution est trouvée alors cette dernière est renvoyée par la fonction incompleteSearch (ligne 6). Si le réseau de contraintes est prouvé insatisfiable, il est alors nécessaire de relâcher des contraintes du WCN. Pour cela, l'algorithme extrait un WCN W' à partir d'un MUC calculé (lignes 8 et 9). La structure $front$ est alors mise à jour par la fonction $relax$ (ligne 10). Cette procédure relâchera une (ou plusieurs contraintes) de W' afin de rendre satisfiable le MUC associé à W'. Ce processus boucle tant qu'une solution n'est pas trouvée au réseau de contraintes associé à W.

La fonction $relax(W, front)$ modifie le tableau $front$ pour autoriser de nouvelles strates. Le principe général est d'incrémenter $front[w]$ pour au moins une contrainte w, de manière à rendre le MUC satisfiable. Autrement dit, pour au moins l'une des contraintes, on s'autorise une augmentation du coût maximal de cette contrainte. Dans l'approche gloutonne adoptée ici, on cherche juste à modifier le tableau $front$ de manière à ce que le MUC soit cassé.

L'algorithme relax utilise une file de priorité locale. La file Q_{loc} est tout d'abord initialisée avec la structure $front$. Tant que Q_{loc} n'est pas vide (ligne 2), on extrait de la liste le front f ayant le coût $cost(f)$ le plus faible. La fonction $\texttt{toCN}_{\leq}$ construit un réseau de contraintes à partir du WCN et du front f précédemment sélectionné (ligne 4). Si ce réseau de contraintes est satisfiable, la structure $front$ passée initialement en paramètre est mise à jour et retournée par l'algorithme relax. Si le réseau de contraintes n'est pas satisfiable, le relâchement n'est pas suffisant et l'algorithme met à jour sa liste de fronts en générant tous les voisins de f. Pour chaque contrainte apparaissant dans le WCN (issu du MUC), on génère une nouvelle structure $front$ différente de la structure initiale par l'incrémentation d'une strate d'une unique contrainte.

On remarquera que, dans les appels à la procédure relax, on peut se restreindre à un sous-ensemble de la structure $front$. En effet les contraintes susceptibles d'être relâchées sont celles faisant partie du WCN W associé au MUC. Pour économiser de l'espace, on peut donc se restreindre en pratique à une structure $front$

Function incompleteSearch(W : WCN)

```
1  foreach w ∈ cons(W) do
2      front[w] ← 0
3  repeat
4      P ← toCN≤(W, front)
5      sol ← solveCN(P)
6      if sol ≠ ⊥ then
7          return sol
8      else
9          M ← extractMUC(P)
10         W' ← restrict(W, M)
11         front ← relax(W', front)
12 until sol ≠ ⊥
```

Function relax(W : WCN, front : array of indexes of levels) : array of indexes of levels

```
1  Q_loc ← {front}
2  while Q_loc ≠ ∅ do
3      f ← pick and delete least cost front in Q_loc
4      P ← toCN≤(W, f)
5      if solveCN(P) ≠ ⊥ then
6          return f
7      else
8          M ← extractMUC(P)
9          foreach w ∈ cons(W, M) do
10             if f[w] ≠ nbLevels(w) − 1 then
11                 f' ← f
12                 f'[w] ← f'[w] + 1
13                 if cost(f') ≠ k then
14                     Q_loc ← Q_loc ∪ {f'}
```

ne contenant pas l'intégralité des contraintes initiales du WCN (comme présentée dans les algorithmes) mais contenant uniquement les contraintes du WCN apparaissant dans W.

Dans la version incomplète, la fonction $\texttt{toCN}_{\leq}$ permettant l'extraction d'un réseau de contraintes à partir d'un front est différente de la fonction $\texttt{toCN}_{=}$ utilisée dans la version complète. En effet, dans la version complète, tous les relâchements possibles d'un MUC sont envisagés et les fronts sont énumérés par ordre de coût croissant, aussi lorsqu'on teste la satisfiabilité d'un CN associé à un front f, on sait que tous les CN associés à des fronts f' tel que $f' \rightarrow_* f$ ont déjà été prouvés insatisfiables. Dans la version complète, la fonction `toCN` extrait donc un CN constitué des strates courantes d'un front. Dans la version incomplète, tous les relâchements possibles d'un MUC ne sont pas envisagés. On ne peut donc pas garantir que lorsqu'on teste la satisfiabilité d'un CN associé à un front f, tous les CN associés à des fronts f' tel que $f' \rightarrow_* f$ ont déjà été prouvés insatisfiables. La fonction `toCN` extrait donc un CN à partir des strates courantes et inférieures d'un front.

Cette approche ne garantit pas l'identification de la solution optimale. D'une part, l'ensemble des relâchements issus des MUC ne sont pas considérés contrairement à la version complète (voir exemple 3). En effet on se contente d'identifier le premier relâchement permettant de restaurer la satisfiabilité des MUC. D'autre part, considérer uniquement le relâchement localement optimal du MUC ne garantit pas d'obtenir la solution optimale du WCN.

6 Résultats expérimentaux

Les expérimentations ont été menées sur un ordinateur équipé de processeurs Intel(R) Core(TM) i7-2820QM CPU 2.30GHz. Nous avons choisi de comparer notre approche gloutonne, notée GMR, à deux approches complètes avec algorithme de transferts de coûts maintenant EDAC proposées par notre solveur AbsCon et par le solveur Toulbar2. À ce stade de notre étude, la version complète présentée dans ce papier est en cours d'implantation dans notre solveur. Le temps limite alloué pour résoudre chaque instance est de 600 secondes. Le temps CPU total pour résoudre chaque instance est donné ainsi que la borne trouvée (UB). Si l'exécution de l'algorithme n'est pas terminée avant le temps limite (CPU supérieur à 600 secondes), la borne affichée correspond à la borne trouvée au bout des 600 secondes. Le tableau 1 représente les résultats obtenus pour la série d'instances *spot*5 constituées de contraintes d'arité au maximum égale à 3.

Nous observons que notre approche donne de meilleures bornes que celles retournées par les deux approches complètes exceptées pour les instances *spot*5-42, *spot*5-404, *spot*5-503 et *spot*5-1502. Au delà du fait que notre approche ne soit pas complète, ceci s'explique par le fait qu'il s'agisse d'instances relativement faciles (moins de 1400 contraintes) et donc les algorithmes à transferts de coûts parviennent à fournir une meilleure borne dans le temps imparti. Cependant, il est intéressant de noter que la borne trouvée par notre approche n'est relativement pas si éloignée de celles trouvées par les approches complètes, et ceci dans un temps très faible. Concernant les autres instances,

		AbsCon		Toulbar2
Instances		GMR	EDAC	EDAC
*spot*5-42	CPU	5.65	> 600	> 600
	UB	162050	**161050**	**161050**
*spot*5-404	CPU	3.17	> 600	217
	UB	118	**114**	**114**
*spot*5-408	CPU	7	> 600	> 600
	UB	**6235**	8238	6240
*spot*5-412	CPU	11.4	> 600	> 600
	UB	**33403**	43390	37399
*spot*5-414	CPU	23.5	> 600	> 600
	UB	**40500**	56492	52492
*spot*5-503	CPU	3.67	> 600	> 600
	UB	12125	13119	**12117**
*spot*5-505	CPU	7.61	> 600	> 600
	UB	**22266**	28258	25268
*spot*5-507	CPU	13.3	> 600	> 600
	UB	**30417**	37429	37420
*spot*5-509	CPU	19.5	> 600	> 600
	UB	**37469**	48475	46477
*spot*5-1401	CPU	45.6	> 600	> 600
	UB	**483110**	513097	516095
*spot*5-1403	CPU	85	> 600	> 600
	UB	**493265**	517260	507265
*spot*5-1407	CPU	334	> 600	> 600
	UB	**495615**	517623	507633
*spot*5-1502	CPU	2.47	0.98	0.02
	UB	28044	**28042**	**28042**
*spot*5-1504	CPU	40.9	> 600	> 600
	UB	**175308**	204314	198318
*spot*5-1506	CPU	207	> 600	> 600
	UB	**388556**	426551	399568

TABLE 1 – Temps CPU en secondes et borne trouvée avant le temps limite pour des instances *spot5*.

considérées comme plus difficiles (notamment plus de 13000 contraintes pour les instances *spot*5-1403, *spot*5-1407 et *spot*5-1506), nous pouvons observer que notre approche gloutonne renvoie une borne meilleure que celles des approches complètes et dans un temps bien plus faible que le temps limite.

7 Conclusion

Dans ce papier, nous proposons une approche originale permettant la résolution d'un réseau de contraintes pondéré (WCN) grâce à la résolution successive de réseaux de contraintes (CNs) dérivés du WCN initial. Plus précisément, les réseaux de contraintes sont énumérés en relâchant un sous-ensemble des contraintes du WCN initial tant que les CNs sont prouvés insatisfiables. Afin d'améliorer la complexité en pratique de notre approche, nous proposons d'identifier un noyau minimalement inconsistant (MUC) pour chaque réseau de contraintes insatisfiable afin de concentrer le relâchement des contraintes du WCN sur les contraintes de ce MUC. L'approche est déclinée en un algorithme complet et un algorithme glouton.

De nombreuses perspectives restent encore à étudier. Tout d'abord, nous avons validé expérimentalement dans ce papier l'algorithme glouton mais il nous reste à valider l'approche complète. Enfin des améliorations sont à envisager dans la méthode d'extraction des MUC. Par exemple, la prise en compte des caractéristiques des contraintes présentes dans le MUC permettrait de mieux cibler celles à relâcher. Ceci pourrait permettre d'éviter l'extraction de nombreux MUC par la suite et par conséquent permettre d'obtenir une borne plus rapidement.

Remerciements

Ce travail bénéficie du soutien du CNRS et d'OSEO dans le cadre du projet ISI Pajero.

Références

[1] C. Ansotegui, M.L. Bonet, and J. Levy. A new algorithm for weighted partial maxsat. In *Proceedings of AAAI'10*, pages 3–8, 2010.

[2] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based CSPs and valued CSPs : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999.

[3] M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3) :311–342, 2003.

[4] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258, 2008.

[5] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8) :449–478, 2010.

[6] S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa. Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. In *Proceedings of IJCAI'05*, pages 84–89, 2005.

[7] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[8] Z. Fu and S. Malik. On solving the partial max-sat problem. In *Proceedings of SAT'06*, pages 252–265, 2006.

[9] F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *Proceedings of ECAI'06*, pages 113–117, 2006.

[10] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[11] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

[12] C. Lecoutre. *Constraint networks : techniques and algorithms*. ISTE/Wiley, 2009.

[13] J.P. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *DATE'08*, pages 408–413, 2008.

[14] P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In *Handbook of Constraint Programming*, chapter 9, pages 281–328. Elsevier, 2006.

[15] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.