Une contrainte Stretch expliquée

Guillaume Rochart, Narendra Jussien

Une contrainte Stretch expliquée

Guillaume Rochart[1,2]
[1]IRIN
2, rue de la Houssinière
F-44322 Nantes
email: grochart@emn.fr

Narendra Jussien[2]
[2]École des Mines de Nantes
4, rue Alfred Kastler
F-44307 Nantes
email: jussien@emn.fr

Résumé

Nous montrons dans cet article comment étendre une contrainte globale, `stretch`, pour l'utiliser dans un contexte *expliqué*. Ceci se traduit notamment par la production d'explications précises pour chacune des décisions prises lors du filtrage. Les expérimentations montrent une amélioration notable de la résolution de problèmes mettant en œuvre une telle contrainte.

1 Introduction

Planifier les emplois du temps des ressources humaines, afin d'améliorer la qualité de service et de rationaliser les dépenses, est le lot quotidien de nombreuses entreprises. Qu'il s'agisse de déterminer les horaires, les jours de congés ou encore les charges de travail des employés, de nombreuses techniques sont proposées. Parmi celles-ci, la programmation par contraintes offre une modélisation de haut-niveau et propose des solutions réutilisables. Ainsi, la contrainte `stretch` a été récemment proposée par [Pesant, 2001] pour prendre en compte des contraintes sur les séquences d'affectations de travail pour les employés de sorte à améliorer la qualité des emplois du temps générés.

L'intérêt des explications (ou *nogoods* pour la programmation par contraintes) a maintenant bien été montré dans les algorithmes Dynamic Backtracking [Ginsberg, 1993], son extension mac-dbt [Jussien *et al.*, 2000] ou encore decision-repair [Jussien et Lhomme, 2002]. Cependant, la fourniture d'explications précises pour les contraintes globales reste un problème mal résolu dans le cas général ([Jussien, 2001; Ågren, 2002]). Dans cet article, nous montrons comment produire des explications précises pour la contrainte globale `stretch` et surtout leur intérêt pour la résolution.

Cet article présente d'abord la contrainte `stretch` en explicitant son utilisation et son mode de fonctionnement, présentés dans [Pesant, 2001]. Nous rappellerons ensuite quelques notions sur les explications et leur utilisation. Puis nous montrerons comment *expliquer* chacune des décisions de filtrage prises par l'algorithme de [Pesant, 2001] afin de pouvoir utiliser la contrainte dans un algorithme exploitant ces informations, comme

mac-dbt. Enfin, nous présenterons une étude des résultats obtenus avec cette nouvelle version de la contrainte.

2 La contrainte `stretch`

La contrainte `stretch` [Pesant, 2001] a été introduite dans le but de déterminer les horaires types d'un ensemble d'employés pour répondre à une demande donnée : c'est ce qu'on appelle une affectation de *shifts*. Elle peut aussi être utilisée pour déterminer les jours de congés (sans prendre en compte des notions de *shifts*), ou encore la répartition des tâches élémentaires dans la journée, soit, plus généralement, toute propriété qui se modélise par une succession de contraintes d'affectations à des activités.

2.1 Un exemple : planning d'infirmières

Organiser un planning d'infirmières consiste à affecter chaque infirmière ou équipe d'infirmières à un certain nombre de créneaux (matin – `M`, soir – `S`, nuit – `N`, ou encore repos – -) de telle sorte que, bien sûr, les contraintes de service soient respectées ainsi que diverses contraintes réglementaires. Il est important, pour assurer une certaine stabilité dans les emplois du temps, de pouvoir préciser un nombre minimal et maximal d'affectations identiques à la suite. Il paraît, par exemple, inconcevable, de travailler de nuit, puis du matin, et enfin du soir, ce qui bouleverserait le cycle de sommeil des employées. Cette contrainte permet alors d'assurer des contraintes du type :

- Un bloc d'affectation du matin (`M`) doit durer entre 3 et 4 jours consécutifs ;
- Un bloc d'affectation du soir (`S`) doit durer entre 3 et 4 jours consécutifs ;
- Un bloc d'affectation de nuit (`N`) doit durer entre 6 et 7 jours consécutifs.

Ayant considéré de plus, qu'à tout moment au moins une équipe est présente, une solution (cf. [Pesant, 2001]) pour un planning concernant cinq équipes serait alors, par exemple :

Semaine	Lun	Mar	Mer	Jeu	Ven	Sam	Dim
1	-	-	-	M	M	M	M
2	-	-	S	S	S	-	-
3	M	M	M	-	-	S	S
4	S	S	-	-	N	N	N
5	N	N	N	N	-	-	-

Tab. 1: Affectation d'horaires pour des infirmières

Cette solution peut alors être utilisée de deux manières :

- Soit les équipes tournent[1], dans ce cas, chaque employée se voit affecté l'emploi du temps d'une semaine i puis à la fin de la semaine change d'emploi du temps pour celui de la semaine $i + 1 \bmod 5$;
- Soit un emploi du temps personnel est calculé, ce qui permet de mettre en place des contraintes personnalisées (réunion, absence pour raison personnelle, ...).

[1] On parle alors de *rostering*.

2.2 Définition de la contrainte

Nous décrivons ici une version légèrement modifiée de la contrainte proposée dans [Pesant, 2001] pour en simplifier la présentation[2]. Supposons que l'on souhaite affecter n jours numérotés de 0 à $n-1$[3], de manière cyclique : au jour n, l'employé est affecté à la même activité qu'au jour 0, etc. Notons x_i l'affectation du jour i. Enfin, notons $\mathcal{D}_{x_i} \subseteq \mathcal{T}$ le domaine de x_i avec $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ l'ensemble des activités possibles.

Définition 1 (Bloc) *Une sous-suite* $x_i, x_{(i+1) \bmod n}, \ldots, x_j$ *est appelé* bloc *si* $x_i = \ldots = x_j$ *mais* $x_{(i-1) \bmod n} \neq x_i$ *et* $x_{(j+1) \bmod n} \neq x_j$. *On appelle* étendue *ou* taille *du bloc comprenant* x_k *(avec* $i \leq k \leq j$*) sa longueur définie par* $etendue(x_k) = 1 + (j - i) \bmod n$.

Soient $\check{\lambda}$ et $\hat{\lambda}$ deux vecteurs de longueur m représentant les étendues minimales et maximales de blocs pour chaque valeur τ_i.

On pose alors la contrainte de la manière suivante :

$$\texttt{stretch}(\langle x_0, x_1, \ldots, x_{n-1} \rangle, \mathcal{T}, \check{\lambda}, \hat{\lambda})$$

avec la sémantique attendue suivante :

- $\forall 0 \leq i \leq n-1$, $x_i \in \mathcal{T}$,
- $\forall 0 \leq i \leq n-1$, $\check{\lambda}_{x_i} \leq etendue(x_i) \leq \hat{\lambda}_{x_i}$,
- $\forall k \in \mathcal{T}$, $\check{\lambda}_k \leq \hat{\lambda}_k$.

Dans le cas des horaires d'infirmière, on pourrait par exemple utiliser la contrainte suivante : $\texttt{stretch}(\langle x_0, \ldots, x_{34} \rangle, \{\texttt{M}, \texttt{S}, \texttt{N}, -\}, \{3, 3, 4, 2\}, \{4, 4, 7, 7\})$

2.3 Propagation

L'implémentation d'une contrainte globale se base principalement sur la propagation (ou algorithme de filtrage) de celle-ci. Nous présentons succinctement ici les bases de ce filtrage tel qu'il est présenté dans [Pesant, 2001] puis nous reviendrons plus tard sur les détails de celui-ci et les modifications que nous lui avons apportées pour l'*expliquer*.

Les algorithmes de filtrage sont basés sur la notion de bloc. À chaque événement (instanciation ou simplement modification d'un domaine), les étendues minimale (c'est-à-dire les variables faisant forcément partie du bloc courant autour d'une variable x_i) et maximale (c'est-à-dire toutes les variables pouvant en faire partie) sont calculées.

Ces étendues minimale et/ou maximale permettent alors de déduire des décisions à prendre (modification de domaine ou signal de contradiction). Par exemple, si l'étendue est forcément trop petite, ou forcément trop grande, une violation de la contrainte peut être déduite. De même, si l'on connaît l'étendue maximale et la longueur minimale d'un bloc, il est possible de déduire des instanciations : par exemple, si l'étendue maximale est de six variables et que la longueur minimale imposée du bloc est de quatre, alors les deux variables centrales appartiennent forcément au bloc ; donc si l'une d'entre elles n'est pas instanciée, une décision d'instanciation peut être prise.

[2] La contrainte originale propose de plus de décrire les patrons de successions autorisées ou non. Par exemple, cela permet d'interdire les horaires trop rapprochés : comme après-midi puis matin, ou oblige d'être en congés après avoir travaillé de nuit. Cependant, comme l'article original le précise, cette propriété n'est pas bien prise en compte par l'algorithme proposé et n'est pas utilisée dans les expérimentations qui nous intéressent ici de [Pesant, 2001].

[3] La notation de 0 à $n-1$ permet de simplifier la notation des modulos.

Cet algorithme de filtrage nous servira de base pour l'implémentation de la contrainte. Cependant, comme nous allons le voir, des explications pour chacune de ces décisions doivent être apportées.

3 Les explications

La résolution de CSP passe bien souvent par un algorithme basé sur le retour arrière chronologique. Les défauts principaux d'une telle approche sont bien connus : absence de mémoire et manque de pertinence dans le retour arrière entraînent un phénomène de *thrashing*.

3.1 Dynamic Backtracking

Pour pallier ces défauts, des solutions à base d'explications ont été proposées dans la littérature, comme backjumping [Kondrak, 1994] ou dynamic backtracking [Ginsberg, 1993].

Une explication donne la raison d'un échec (comme les *nogoods* proposés par [Frost et Dechter, 1994; Schiex et Verfaillie, 1994]) ou d'une inférence (une décision prise lors de la propagation). Ceci se traduit par la détermination lors d'une inconsistance, par exemple, d'un ensemble de choix (pris par l'algorithme de recherche, c'est-à-dire des instanciations dans le cas de contraintes discrètes comme `stretch`) et de contraintes entraînant cette inconsistance.

Définition 2 (Cause d'une inférence) *La cause d'une contradiction est un sous-ensemble inconsistant de contraintes du système ($\mathcal{C}' \subset \mathcal{C}$) et d'instanciations de variables (que l'on peut considérer comme des contraintes de choix, d_1, d_2,..., d_k) : $\mathcal{C}' \wedge d_1 \wedge \ldots \wedge d_n$.*

Plus généralement, la cause d'une inférence $\mathcal{X}$ (contradiction, filtrage, ...) est un sous-ensemble de contraintes (du système et de décision), tel que : $\mathcal{C}' \wedge d_1 \wedge \ldots \wedge d_n \Rightarrow \mathcal{X}$. On notera $expl(\mathcal{X})$ la partie gauche de cette implication.

On peut alors spécialiser cette définition pour les retraits de valeurs dans les domaines d'une variable :

Définition 3 (Explication d'un retrait de valeur) *L'explication d'un retrait de valeur v d'un domaine de variable x est un sous-ensemble de contraintes (du système et de décision) tel que : $\mathcal{C}' \wedge d_1 \wedge \ldots \wedge d_n \Rightarrow x \neq v$.*

Grâce à cette information maintenue par le solveur, l'algorithme dynamic backtracking peut ainsi savoir quelle est la dernière décision en cause pour ce conflit (ce peut être une décision autre que la dernière ayant été prise). L'algorithme propose alors en quelque sorte de modifier l'ordre d'instanciation des variables de sorte à changer cette décision sans revenir sur les autres décisions qui ont été prises depuis, puisqu'elles ne sont pas responsables de l'inconsistance.

3.2 L'algorithme mac-dbt

3.2.1 Principes de l'algorithme

Un défaut de dynamic backtracking est qu'il ne permet pas de prendre en compte des techniques à base de maintien d'arc consistance ou de propagation de contraintes.

L'idée proposée par [Jussien *et al.*, 2000] revient à utiliser le maintien d'arc-consistance (mac) au sein du dynamic backtracking (dbt). Ce nouvel algorithme, mac-dbt, a pour principale contrainte de devoir expliquer non seulement chacune des contradictions comme dynamic backtracking mais aussi de devoir expliquer chacune des réductions de domaines inférées lors de la propagation.

Exemple 1 *Par exemple, considérons trois variables* $x_1, x_2, x_3 \in [\![1, 4]\!]$ *avec les contraintes* $c_1 = x_1 < x_2$ *et* $c_2 = x_2 < x_3$*. Dans ce cas, une première propagation aura lieu donnant par exemple lieu au retrait de* 4 *dans* x_1*. Ce retrait ne dépendant que de la contrainte* c_1*, le domaine de* x_1 *stockera non seulement le retrait de* 4 *du domaine mais aussi l'explication* $expl(x_1 \neq 4) = \{c_1\}$.

De même, la variable stockera l'explication suivante : $expl(x_1 \neq 3) = \{c_1, c_2\}$ *puisque la conjonction de ces deux contraintes permettent bien de déduire un tel retrait. Dés lors, si une contradiction est déduite du fait que* 3 *n'est plus dans le domaine de* x_1*, on pourra très simplement déduire qu'il faut soit retirer la contrainte* c_1 *soit retirer la contrainte* c_2*. C'est ce principe qui est utilisé par* mac-dbt *pour améliorer la recherche.*

3.2.2 Implémentation de l'algorithme

Pour implémenter un tel algorithme, il fait donc fournir à chaque retrait de valeur une explication justifiant ce dernier. Par exemple, dans le cas d'une contrainte d'infériorité (la contrainte c_1 de l'exemple ci-dessus), la modification de la borne inférieure de la première variable x_1 induit une modification sur la variable de droite x_2. Dans ce cas, cette inférence dépend d'une part de la contrainte elle-même et d'autre part des retraits des valeurs de x_1.

De manière générale, si l'on se restreint aux CSPs binaires où les contraintes sont exprimées sous la forme de tuples de valeurs authorisés, l'application de la contrainte c_{xy} sur x et y, une valeur v de x est retirée si toutes les valeurs supportant $x = v$ dans y sont retirées. On a donc :

$$expl(x \neq v) = c_{xy} \cup \bigcup_{b \in \mathcal{D}_y \text{ support de } x=v} expl(y \neq b)$$

Dans le cas d'une contrainte globale, il n'est pas envisageable de générer les supports pour chaque variable, il faut donc tirer partie des algorithmes de filtrage évolués qui sont fournis avec de telles contraintes. Ainsi, pour pouvoir utiliser des contraintes globales (comme `stretch`) dans un tel contexte, il faut les modifier pour expliquer chacune des décisions prises lors du filtrage, que ce soit lors d'un retrait de valeur ou d'une inconsistance. Or ces décisions dépendent souvent de calculs complexes et de structures, internes à la contrainte, qui obligent à modifier le code de la propagation de la contrainte pour pouvoir expliquer ces décisions.

Dans le cas de `stretch`, nous allons donc reprendre l'implémentation de `stretch` proposée par [Pesant, 2001] en proposant pour chaque calcul une *explication* des décisions prises. Ceci se traduit par le calcul des causes aussi précises que possible dans le calcul des bornes (*quels sont les choix passés justifiant la valeur d'une borne ?*), et la déduction d'explications pour chacune des décisions prises par l'algorithme de filtrage.

4 Calcul expliqué des bornes des blocs

Comme nous l'avons vu lors la présentation de la contrainte, le filtrage de la contrainte `stretch` dépend principalement du calcul des bornes potentielles du bloc courant (c'est-à-dire, le bloc englobant la variable étudiée). Nous allons donc rappeler la méthode de calcul proposée par [Pesant, 2001], et présenterons les causes et explications que nous proposons.

Les blocs sont représentés par quatre valeurs : β_{min} la valeur minimale de début du bloc, β_{max} la valeur maximale de début du bloc, et ϵ_{min} et ϵ_{max} définies de manière identique pour la fin du bloc comme l'indique la figure 1.

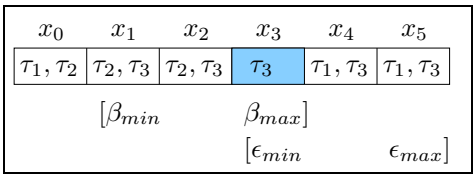


FIG. 1: Modélisation des blocs pour l'étude de x_3

Nous supposerons dans l'ensemble de l'article que x_i représente la variable d'étude, et τ_k la valeur du bloc étudié.

4.1 Calcul de β_{max}

Commençons tout d'abord par définir de manière formelle la borne β_{max}[4].

Définition 4 (Borne β_{max}) *β_{max} est définie telle que :*
- $\forall p \in [\![\beta_{max}, i]\!], \mathcal{D}_{x_p} = \{\tau_k\}$
- $\mathcal{D}_{x_{(\beta_{max}-1) \bmod n}} \neq \{\tau_k\}$

Le calcul de la valeur maximale de la borne inférieure exploite le fait que toutes les variables voisines instanciées à la même valeur font forcément partie du même bloc. Ainsi, β_{max} et ϵ_{min} permettront de connaître l'étendue certaine du bloc courant.

L'algorithme de recherche proposé par [Pesant, 2001] (figure 2) parcourt donc les variables précédant la variable d'étude dans l'ordre de la suite (figure 3).

```
début
    j ← (i − 1) mod n
    tant que x_j = τ_k faire
        j ← (j − 1) mod n
    fintantque
    β_max ← (j + 1) mod n
fin
```

FIG. 2: Alg. de calcul de β_{max}

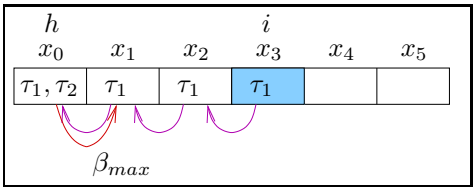


FIG. 3: Calcul de β_{max}

[4] ϵ_{min} sera défini d'une manière similaire. En effet, les calculs des bornes sont symétriques : β_{max} et ϵ_{min} sont toutes les deux des bornes permettant de savoir l'étendue certaine du bloc en cours. Il en sera de même pour β_{min} et ϵ_{max}.

Afin d'expliquer chacune des décisions prises lors de la propagation, il faut pouvoir justifier chacun des calculs, et donc déterminer la cause de cette valeur de β_{max}. Dans la pratique, ce n'est pas la valeur de β_{max} qui nous intéresse mais un majorant de cette borne. En effet, les règles de propagation qui seront utilisées seront basées sur le fait que le bloc s'étend au moins jusqu'à cette limite (étendue supérieure à une valeur, $\beta_{max} = \beta_{min}$, ...), même s'il est évident que plus la valeur est petite modulo n, plus la propagation est efficace.

Dans notre cas, le fait que β_{max} soit au niveau de x_1 ($\beta_{max} = 1$) s'explique par les domaines de x_1, x_2 et x_3 mais aussi par celui de x_0. Mais puisque l'on souhaite juste expliquer que la borne se trouve soit en x_1 soit avant ($\beta_{max} \leq 1$) dans l'ordre de la suite, la cause suivante suffit :

$$cause(\beta_{max} \leq 1) = expl(\mathcal{D}_{x_3}) \cup expl(\mathcal{D}_{x_2}) \cup expl(\mathcal{D}_{x_1})$$

où $expl(\mathcal{D}_{x_i})$ représente l'explication de l'état actuel du domaine, c'est-à-dire l'union des explications de chaque retrait de valeur : $expl(\mathcal{D}_{x_i}) = \bigcup_{r \in \mathcal{T}, x_i \neq r} expl(x_i \neq r)$

De manière générique, nous aurons donc la cause :

Définition de cause 1 (β_{max})

$$cause(\beta_{max} \leq h) = \bigcup_{m \in [\![h,i]\!]} expl(\mathcal{D}_{x_m})$$

Idée de preuve

Supposons que l'on ait $\beta_{max} > h + 1$, cela supposerait qu'il est possible qu'il y ait un bloc de valeur différente de τ_k dans la sous-suite restreinte à $x_{h+1}, \ldots, x_{\beta_{max}-1}$, c'est-à-dire qu'au moins une variable contient dans son domaine une valeur différente de τ_k. Or la cause contient justement l'explication des domaines de chacune de ces variables.

❑

4.2 Calcul de β_{min}

Cette borne permet de déterminer assez précisément l'ensemble des variables pouvant appartenir au bloc courant de τ_k. Pour cela, la définition suivante lui a été donnée :

Définition 5 (Borne β_{min}) *β_{min} est définie telle que :*
- $\forall p \in [\![\beta_{min}, \beta_{max}]\!], \tau_k \in \mathcal{D}_{x_p}$,
- $\exists l, \tau_l \neq \tau_k, \forall p \in [\![(\beta_{min} - \check{\lambda}_l) \bmod n, (\beta_{min} - 1) \bmod n]\!], \tau_l \in \mathcal{D}_{x_p}$, *c'est-à-dire qu'un autre bloc peut exister à la frontière.*

Le calcul de la valeur minimale de la borne inférieure se déroule en deux étapes :
- Calcul d'une borne grossière (notée β_{min}^{prov}), ce qui reviendra à se limiter aux variables pouvant être instanciées à τ_k ;
- Calcul de la véritable borne en vérifiant qu'un bloc voisin est viable.

Nous allons donc expliquer chacune de ces étapes séparément de sorte à pouvoir utiliser l'algorithme proposé par [Pesant, 2001].

4.2.1 Calcul d'une borne grossière (ou provisoire)

Le calcul de la borne grossière est relativement simple. Les variables précédant la variable de travail sont parcourues jusqu'à ce que :

– Soit une variable non instanciable à τ_k est atteinte ;

– Soit on est trop loin de ϵ_{min} étant donnée la longueur maximale autorisée.

L'algorithme proposé par [Pesant, 2001] parcourt alors les variables jusqu'à ce que l'une des deux conditions soit vérifiée. De plus, il prend en compte le fait que deux blocs de même valeur ne peuvent se suivre comme indiqué dans la figure 4.

```
1   début
2     si λ̂_k ≥ n alors
3       troploin ← (ε_min + 1) mod n
4     sinon
5       troploin ← (ε_min − λ̂_k) mod n
6     finsi
7     j ← (β_max − 1) mod n
8     fini ← faux
9     tant que j ≠ troploin ∧ non(fini) faire
10      tant que τ_k ∈ D_{x_j} ∧ |D_{x_j}| > 1 ∧ j ≠ troploin faire
11        j ← (j − 1) mod n
12      fintantque
13      si τ_k ∉ D_{x_j} alors
14        β_min^prov ← (j + 1) mod n      % cas a
15        fini ← vrai
16      sinon
17        si |D_{x_j}| = 1 alors
18          j' ← j
19          tant que x_j = τ_k ∧ j ≠ troploin faire
20            j ← (j − 1) mod n
21          fintantque
22          si x_j = τ_k alors
23            β_min^prov ← (j' + 2) mod n      % cas c
24            fini ← vrai
25          finsi
26        finsi
27      finsi
28    fintantque
29    si non(fini) alors
30      β_min^prov ← (j + 1) mod n      % cas b
31    finsi
32  fin
```

FIG. 4: Calcul d'une valeur grossière de β_{min}

Il y a donc trois cas d'arrêt à traiter.

a – Une variable non instanciable à τ_k est atteinte (ligne 14) : Dans ce cas, la valeur de β_{min} se justifie simplement par l'impossibilité d'instancier cette variable à

la valeur τ_k :

Définition de cause 2 ($\beta^{prov}_{min} - \tau_k \notin \mathcal{D}_{x_h}$)

$$cause(\beta^{prov}_{min} \geq h+1) = expl(x_h \neq \tau_k) \cup expl(\mathcal{D}_{x_i})$$

Idée de preuve

Supposons que β^{prov}_{min} soit plus petit modulo n. Dans ce cas, on aurait toutes les variables entre cette borne et x_i qui seraient instanciables à τ_k, soit $\tau_k \in \mathcal{D}_{x_h}$. Or la cause contient justement la justification de $\tau_k \notin \mathcal{D}_{x_h}$.

❑

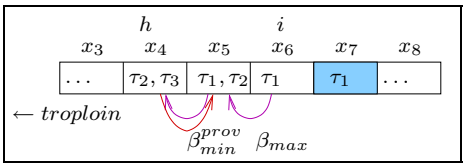


FIG. 5: Exemple de calcul de β^{prov}_{min} (cas 1)

Exemple 2 *Supposons le cas de la figure ci-contre.*
La cause serait alors :

$$cause(\beta^{prov}_{min} \geq 5) = expl(x_4 \neq \tau_1) \cup expl(\mathcal{D}_{x_7})$$

b – La recherche est trop éloignée et la variable atteinte n'est pas instanciée à la même valeur (ligne 30) : Dans ce cas, la valeur dépend de l'emplacement de la fin du bloc, c'est-à-dire de ϵ_{min} :

Définition de cause 3 ($\beta^{prov}_{min} - troploin \wedge \mathcal{D}_{x_h} \neq \{\tau_k\}$)

$$cause(\beta^{prov}_{min} \geq h+1) = expl(\epsilon_{min} \geq g) \cup expl(\mathcal{D}_{x_i})$$

où $expl(x \geq v) = \cup_{i<v} expl(x \neq i)$.

c – La recherche est trop éloignée et la variable atteinte fait forcément partie d'un bloc de même valeur (ligne 23) : Ce cas est un peu plus complexe. Non seulement ϵ_{min} contraint la localisation de β_{min} via la longueur maximale du bloc, mais en plus, la valeur de la variable atteinte et des précédentes (dans l'ordre du parcours) ayant la même valeur τ_k implique un décalage de sorte à éviter d'avoir deux blocs similaires de suite.

Définition de cause 4 ($\beta^{prov}_{min} - troploin \wedge \mathcal{D}_{x_h} = \{\tau_k\}$)

$$cause(\beta^{prov}_{min} \geq h'+2) = expl(\epsilon_{min} \geq g) \cup expl(\mathcal{D}_{x_i}) \cup \bigcup_{m \in [\![troploin, h']\!]} expl(\mathcal{D}_{x_m})$$

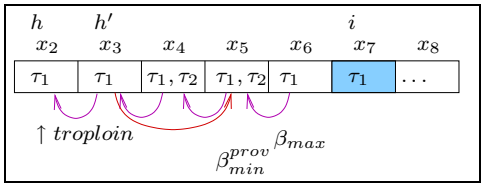


FIG. 6: Exemple de calcul de β^{prov}_{min} (cas 3)

Exemple 3 *Supposons le cas de la figure ci-contre.*
La cause serait :

$$cause(\beta^{prov}_{min} \geq 5) = expl(\epsilon_{min}) \cup expl(\mathcal{D}_{x_7}) \cup expl(\mathcal{D}_{x_2}) \cup expl(\mathcal{D}_{x_3})$$

4.2.2 Calcul de la borne finale

Grâce à cette borne provisoire, il est possible d'affiner la valeur de β_{min} en vérifiant maintenant que les blocs précédents sont viables (figure 7).

```
début
  j ← (β_min^prov − 1) mod n
  tant que j ≠ β_max faire
    pour tout τ_l ∈ D_{x_j} tel que τ_l ≠ τ_k faire
      trouve ← vrai
      pour i de 1 à λ̆_l − 1 faire
        si τ_l ∉ D_{x_{(j−i) mod n}} alors
          trouve ← faux
        finsi
      finpour
      si trouve alors
        retourner (j + 1) mod n
      finsi
    finpour
    j ← (j + 1) mod n
  fintantque
  retourner −1
fin
```

FIG. 7: Calcul de la valeur finale de β_{min}

L'algorithme peut être utilisé directement pour expliquer la nouvelle borne. Chaque décalage ($j \leftarrow j+1 \bmod n$) est dû au fait que *trouve* soit faux, c'est-à-dire à l'impossibilité d'instancier certaines variables à une valeur donnée. Donc la cause du ou des décalages est simplement l'explication de l'absence de ces valeurs testées dans les domaines. De plus, l'explication du domaine de la première variable étudiée est ajoutée pour justifier la longueur de la recherche. En notant α la borne provisoire, la cause est alors[5] (avec i, j, et l prenant les valeurs comme définies dans l'algorithme de la figure 7) :

$$cause(\beta_{min} \geq \alpha + n) \quad = \quad cause(\beta_{min}^{prov} \geq \alpha) \cup \bigcup_{j} expl(\mathcal{D}_{x_j}) \cup \bigcup_{i,j,l} expl(\tau_l \notin \mathcal{D}_{x_{(j-i) \bmod n}})$$

[5]On notera que cette explication (ainsi que celles introduites précédemment) reste valable dans le cas de la contrainte `stretch` telle qu'elle était présentée par [Pesant, 2001], puisque les éventuelles contraintes de successions sont intrinsèques à la contrainte et donc ne dépendent en aucune façon de l'état des variables.

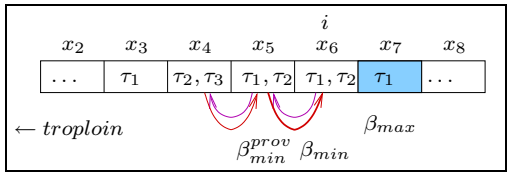


FIG. 8: Exemple de calcul de β_{min}

Exemple 4 *Si l'on considère le cas ci-dessus avec* $\check{\lambda}_2 = 2$ *et* $\check{\lambda}_3 \geq 2$*, l'explication est :*

$$cause(\beta_{min} \geq 6) \quad = \quad \overbrace{expl(x_4 \neq \tau_1) \cup expl(\mathcal{D}_{x_7})}^{cause(\beta_{min}^{prov} \geq 5)} \cup \underbrace{expl(\mathcal{D}_{x_4}) \cup expl(x_3 \neq \tau_2) \cup expl(x_3 \neq \tau_3)}_{\textit{cause de l'unique décalage}}$$

Toutes ces explications de bornes vont permettre maintenant d'expliquer chacune des décisions qui peuvent être prises lors de la propagation de la contrainte. Nous allons donc désormais décrire de manière sommaire les différentes règles de propagation et leurs explications associées.

5 Les règles de propagation avec explications

Afin d'expliciter l'utilisation de cette contrainte, nous allons présenter les différentes règles à travers un exemple. Supposons que nous ayons une suite de 10 variables pouvant prendre des valeurs entre 1 et 3. De plus, supposons que nous ayons les contraintes suivantes sur les longueurs de blocs :

Valeur du bloc (τ)	1	2	3
Longueur minimale ($\check{\lambda}$)	1	2	3
Longueur maximale ($\hat{\lambda}$)	2	3	4

TAB. 2: Longueurs autorisées pour les blocs

De plus, on notera maintenant les explications sous la forme $expl(\beta_{min})$ et non plus sous la forme d'inégalité, car il n'y a pas de problème d'interprétation.

➠ *Supposons qu'un algorithme de recherche prenne les décisions suivantes :*

❶ $x_5 = 1$ *: il est difficile d'en déduire quelque chose*

❷ $x_7 \neq 3$ *: on peut en déduire que* $x_6 \neq 3$

En effet, les blocs de 3 doivent avoir une longueur de 3 au minimum. Donc, puisque $x_5 \neq 3$ et $x_7 \neq 3$, il est clair que $x_6 \neq 3$.

Règle F1

Lors du retrait d'une valeur d'un domaine, on applique l'algorithme suivant :

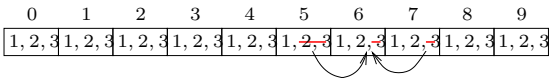
```
début
  pour tout τ_l qui vient d'être retiré de D_{x_i} faire
    si τ_l ∈ D_{x_{(i-1) mod n}} alors
      j ← calcul du β_min d'un bloc potentiel
          de τ_l finissant à (i − 1) mod n
      si longueur(j, (i − 1) mod n) < λ̌_l ou j = −1 alors
        remove(τ_l, D_{x_{(i-1) mod n}})
      finsi
    finsi
  finpour
fin
```

On utilise bien entendu le même algorithme pour les bornes supérieures.
L'explication sera alors, comme l'algorithme l'indique (le retrait dépend uniquement de la valeur de β_{min} et du retrait qui vient d'être propagé) :

$$expl(x_{(i-1) \bmod n} \neq \tau_l) = cause(\beta_{min}) \cup expl(x_i \neq \tau_l)$$

Dans notre exemple, cela se traduit par :

$$expl(x_6 \neq 3) = \underbrace{expl(x_5 \neq 3) \cup expl(\mathcal{D}_{x_6})}_{cause(\beta_{min})} \cup expl(x_7 \neq 3)$$



➠ *Supposons maintenant que l'algorithme de recherche ajoute la décision suivante :*

❸ $x_4 = 1$ *: le bloc de 1 est complètement connu, car sa taille est limitée à 2*

On peut donc en déduire que les blocs voisins ne sont pas des blocs de 1, et donc retirer cette valeur des variables voisines.

Règle F5/F6

Si $\beta_{min} = \beta_{max}$ ou $\epsilon_{min} = \epsilon_{max}$, il est possible de retirer τ_k des domaines des variables voisines sur une longueur égale à la plus petite des longueurs minimales des blocs voisins potentiels.
L'explication est alors (dans le cas des bornes inférieures) :

$$\begin{aligned} expl(x_{(i-1) \bmod n} \neq \tau_k) \quad = \quad & cause(\beta_{min}) \cup cause(\beta_{max}) \\ & \cup expl(\mathcal{D}_{x_{(i-1) \bmod n}}) \end{aligned}$$

Dans notre exemple, cela se traduit par :

$$expl(x_3 \neq 1) = expl(\mathcal{D}_{x_4}) \cup expl(\mathcal{D}_{x_5}) \cup expl(\mathcal{D}_{x_3})$$

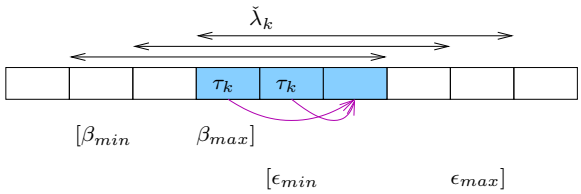
➠ *De même supposons,*

❹ $x_9 = 3$ *: il y aura un bloc de 3 commençant en* x_8 *ou* x_9

Donc on peut en déduire que $x_0 = \{3\}$ puisque la longueur minimale est de 3 : le bloc va soit de x_8 à au moins x_0, soit de x_9 à au moins x_1.

Règle F7

Connaissant β_{min} et ϵ_{max}, si la longueur est suffisante, certaines zones sont forcément couvertes par le bloc :



L'explication sera alors tout simplement l'explication de β_{min} et ϵ_{max} :

$$expl(x_m = \{\tau_k\}) = cause(\beta_{min}) \cup cause(\epsilon_{max})$$

➠ *Enfin,*

❺ $x_3 = 3$ *: ceci implique un bloc d'au moins 3 variables de suite.*

D'où, $x_1 = x_2 = x_3 = 3$ car $x_4 = 1$. Il y a donc un problème, puisque dans ce cas, la longueur du bloc sera au moins 5 car $x_0 = x_9 = \{3\}$. Il faut signaler une contradiction. Pour cela une règle est déclenchée si la longueur sera trop courte ou trop longue (dans ce cas, trop court, car β_{min} vaudra 3 justement pour éviter d'être voisin à un bloc de même valeur).

Règle F4/F3

Si $longueur(\beta_{max}, \epsilon_{min}) > \hat{\lambda}_k$ ou $longueur(\beta_{min}, \epsilon_{max}) < \check{\lambda}_k$ [a], la contrainte est violée.
L'explication sera alors l'union des causes de β_{max},ϵ_{min} ou β_{min},ϵ_{max} respectivement.

[a] On notera que pour l'implémentation, il est plus judicieux de calculer la somme de la longueur entre β_{min} et la variable locale et celle entre cette variable et ϵ_{max}, pour éviter tout problème dû à des longueurs supérieures à la taille de la suite n.

➠ *On aurait pu aussi imaginer la même décision mais avec auparavant une autre décision :*

❻ $x_1 = 3$ *et* $x_3 = 3$

Il est clair que cette fois-ci l'affectation $x_3 = 3$ ne sera pas plus valide, mais la raison sera un peu différente. En effet, dans ce cas, le calcul de β_{min} renverra -1, car $\beta_{min} > 2$ puisque $x_9 = x_0 = x_1 = 3$, $\beta_{min} > 3$ car il n'existe pas de blocs voisins viables : la seule valeur autorisée aurait été un bloc de 2 mais il en faudrait deux consécutivement et $x_1 \neq 2$. D'où la règle suivante :

Règle F2

Si $\beta_{min} = -1$ alors il faut lever une contradiction, la contrainte étant violée. L'explication sera alors tout simplement la cause de β_{min}.

6 Évaluation de la technique

Nos tests ont été implémentés en Choco, moteur à événements de résolution de CSPs [Laburthe, 2000], et PaLM, son extension aux explications [Jussien, 2001].

Plusieurs versions de la contrainte ont été implémentées pour pouvoir tester l'utilisation des explications : une **version classique**, sous la forme d'une contrainte Choco, qui a pour but de refléter la contrainte telle qu'elle a été présentée par [Pesant, 2001] ; une **version expliquée naïvement**, c'est-à-dire une version utilisant la plateforme expliquée mais où les explications sont naïves : l'explication d'une décision (retrait, contrainte violée) est l'ensemble des choix pris sur les domaines des variables de la contrainte ; une **version expliquée améliorée**, c'est-à-dire une version proposant les explications présentées dans cet article. On notera que cette version de la contrainte `stretch` n'est pas plus coûteuse que la version naïve (puisque toutes les modifications sont en $O(1)$) et présente des complexités au pire cas en $O(n.m.e)$ pour le stockage des explications des variables d'une instance de la contrainte (avec e le nombre total de contraintes : les contraintes systèmes et le nombre maximal de contraintes d'énumération – n).

Nous testons ces contraintes sur deux types de problèmes tests : l'un comportant uniquement cette contrainte, et un autre comprenant deux fois cette contrainte (afin de mettre en avant l'influence des explications sur la coopération entre les contraintes).

6.1 Un problème monocontraint

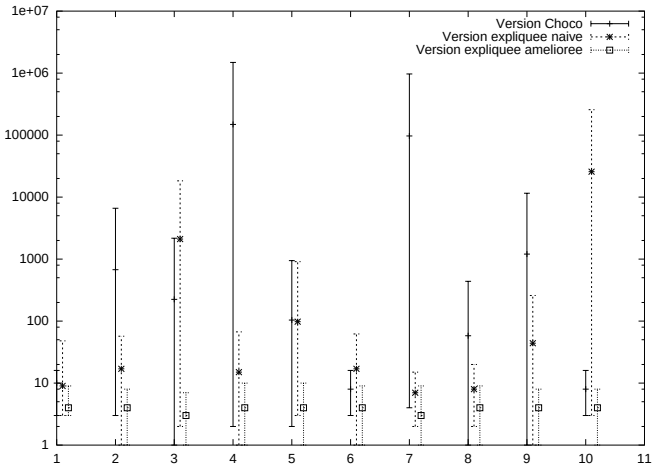


Fig. 9: Évaluation de la contrainte en nombre de backtrack en fonction des instances du problème et des différentes implémentations.

Pour tester ces contraintes, nous nous sommes tout d'abord intéressé aux tests présentés par [Pesant, 2001]. Il s'agit de tester la résolution de cette contrainte sur plusieurs instances étant donné une longueur donnée, un ensemble de valeurs ainsi que des longueurs minimales et maximales pour chaque valeur possible. De plus, comme dans l'article original, l'affectation se fait dans un ordre aléatoire pour simuler la résolution d'un problème complexe[6].

Afin de faciliter la comparaison, nous montrons ici les résultats sur des instances utilisées dans l'article original, pour le cas d'une longueur de 50 variables, pour 7 valeurs et un écart moyen de 5 : il s'agit du cas le plus complexe qui y est abordé.

Les résultats en nombre de *backtracks* pour ces instances sont résumés sur le graphique 9 représentant les valeurs moyennes, minimales et maximales.

On peut tirer deux remarques principales de ces résultats :

- La version **expliquée améliorée** est beaucoup plus stable que les autres versions, avec un nombre de backtracks dans le pire cas souvent largement inférieur à la valeur moyenne des autres versions. Ceci peut s'expliquer, entre autre, par le fait que des explications détaillées permettent de déterminer précisément la décision en cause et donc d'éviter un *trashing* inutile. Ceci permet donc de pallier l'imperfection de la propagation.
- La version **expliquée naïve** semble être un peu meilleure en moyenne que la version **classique**. Ceci peut être dû en partie à l'apport même minime des explications dans le cas où une décision n'implique aucune modification de domaine et donc où l'on pourra éviter de retirer cette décision inutilement. De plus, l'ordre des propagations n'étant pas exactement le même et la propagation n'étant pas parfaite (certaines décisions pouvant être inférées indirectement ne sont pas prises), il est aussi possible que la différence provienne de la phase de propagation.

En ce qui concerne le temps de calcul, le temps moyen de résolution est de 2500 ms pour la version Choco, 9370 ms pour la version expliquée naïvement et de 59 ms pour la version expliquée améliorée. Ces résultats indiquent d'une part que l'utilisation d'explications *précises* permet de largement améliorer la résolution d'un problème mais aussi qu'elle permet d'amortir rapidement les coûts de traitement des explications. Notons que les médianes des temps obtenus se situent à 17.5 et 57.5 pour Choco et la version expliquée améliorée respectivement. Ce résultat montre l'excellente stabilité de cette dernière comparée à la version Choco.

6.2 Un problème bicontraint

Afin de refléter une utilisation plus réaliste de la contrainte, nous avons effectué des tests mettant en œuvre deux contraintes `stretch` entrecroisées sur la moitié des variables du problème. On peut ainsi mettre en évidence l'intérêt d'explications précises pour la coopération de contraintes globales. Pour cela, des tests ont été effectués sur des longueurs comprises entre 25 et 100 (sur une vingtaine d'instances pour en faire une moyenne) avec les trois versions : `choco`, explications naïves et explications précises.

Les résultats présentés figure 10 indiquent une nette diminution à la fois du temps de calcul et du nombre de backtracks nécessaires pour résoudre ces problèmes. En effet,

[6] On notera que des tests montrent que dans le pire cas où l'ordre est intelligent et non modifié par d'autres contraintes, la perte de la version expliquée améliorée par rapport à la version classique est de moins d'un facteur constant 10, résultat confirmant [Jussien *et al.*, 2000].

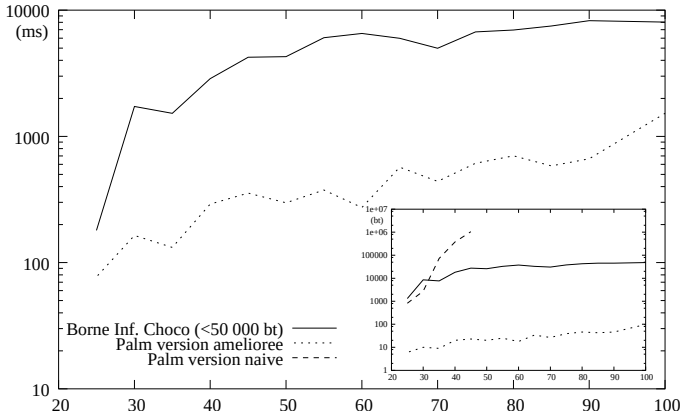


FIG. 10: Comparaison des versions `choco`, explications naïves, explications précises (en temps et en nombre de backtracks) en fonction du nombre de variables en jeu. Les résultats pour la version `choco` sont une **borne inférieure** des valeurs réelles obtenue en limitant la résolution à 50 000 backtracks.

dans les versions `choco` et explications naïves, il n'y aucun partage d'information entre les deux contraintes qui travaillent *en aveugle* et donc conduisent à de nombreux backtracks inutiles ; tandis que la version avec explications précises permet de rapidement identifier les sous-ensembles de variables posant problème entre les deux contraintes et de se focaliser dessus, permettant ainsi une amélioration sensible des performances.

7 Conclusion

La modification de cette contrainte `stretch` pour l'étendre à une plateforme expliquée nous a donc permis de mettre en évidence le gain que peut apporter l'utilisation des explications précises pour la résolution de contraintes globales. En effet, grâce à l'ajout d'explications de chacune des décisions prises lors du filtrage, la résolution de cette contrainte devient beaucoup plus stable en proposant de très bons temps d'exécution au pire cas.

Des travaux similaires sur d'autres contraintes globales ont permis de mettre en avant les différents difficultés que peut présenter l'ajout d'explication dans le cas général des contraintes globales [Rochart *et al.*, 2003] : nécessité d'algorithmes *compatibles* avec les explications, incrémentalité de l'algorithme, calculs efficaces des explications, etc. De plus, des travaux seront menés pour tenter de fournir un cadre général de développement de contraintes globales expliquées. De telles explications pourraient aussi être utilisées dans un cadre coopératif pour aider la collaboration entre plusieurs *solveurs* de contraintes (discrets, continus, simplex, symbolique, ...).

Enfin, nous avons montré ici l'intérêt des explications pour résoudre un problème de satisfaction de contraintes. Cependant, ces explications peuvent être utilisées dans un cadre plus large pour permettre au développeur de déboguer une application grâce à l'information retournée par la contrainte, ou d'une manière générale pour toute interaction avec l'utilisateur [Rochart *et al.*, 2003] : contraintes contradictoires, documentation sur le filtrage...

Références

[Ågren, 2002] Magnus Ågren. Tracing and explaining the execution of clp(fd) programs in sicstus prolog. Master's thesis, Computing Science Department, Uppsala University, Sweden, July 2002.

[Frost et Dechter, 1994] Daniel Frost et Rina Dechter. Dead-end driven learning. Dans *Proceedings of the Twelfth National Conference of Artificial Intelligence (AAAI-94)*. AAAI Press, 1994.

[Ginsberg, 1993] Matthew Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[Jussien *et al.*, 2000] Narendra Jussien, Romuald Debruyne, et Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. Dans *Principles and Practice of Constraint Programming (CP 2000)*, LNCS. Springer-Verlag, 2000.

[Jussien et Lhomme, 2002] Narendra Jussien et Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 2002.

[Jussien, 2001] Narendra Jussien. e-constraints : explanation-based constraint programming. Dans *CP01 Workshop on User-Interaction in Constraint Satisfaction*, 2001.

[Kondrak, 1994] Grzegorz Kondrak. A theoretical evaluation of selected backtracking algorithms. Master's thesis, University of Alberta, 1994.

[Laburthe, 2000] François Laburthe. Choco : implementing a cp kernel. Dans *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore, September 2000.

[Pesant, 2001] Gilles Pesant. A filtering algorithm for the stretch constraint. Dans *Principles and Practice of Constraint Programming (CP 2001)*, number 2239 in Lecture Notes in Computer Science, pages 183–195. Springer-Verlag, 2001.

[Rochart *et al.*, 2003] Guillaume Rochart, Narendra Jussien, et François Laburthe. Challenging explanations for global constraints. Dans *CP03 Workshop on User-Interaction in Constraint Satisfaction*, 2003.

[Schiex et Verfaillie, 1994] Thomas Schiex et Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *International Journal on Artificial Intelligence Tools (IJAIT)*, 3, 1994.