

# Tutoriel : Expliquer une contrainte

by Guillaume Rochart

## Table of contents

1 Construction d'une contrainte expliquée.....	2
2 Propagation sur la contrainte.....	2
3 Propagation sur les variables.....	3
4 Propagation de la restauration.....	4
5 Quelques méthodes utiles.....	5

L'objectif de ce tutoriel est de présenter comment expliquer une contrainte c'est-à-dire comment justifier chacune des décisions de filtrage prises par une contrainte. Ceci permet par la suite d'utiliser la contrainte avec des algorithmes dédiés mais aussi d'avoir des explications sur le comportement du solveur (absence de solutions notamment).

Pour cela, nous montrerons comment implémenter une contrainte de supériorité permettant d'assurer qu'une variable  $x$  est inférieure à une variable  $y$ . Puis nous monterons comment cela se traduit dans le cas de contraintes plus complexes comme les combinaisons linéaires.

## 1. Construction d'une contrainte expliquée

Choco propose plusieurs classes abstraites dont les contraintes que l'on crée peuvent hériter pour éviter de ne devoir coder des méthodes *non métier*. Dans notre cas, on souhaite modéliser la contrainte  $x \# y + c$ . Il s'agit donc d'une classe binaire (deux variables) qui héritera de la classe `AbstractPalmBinIntConstraint`. Pour construire la contrainte il ne reste plus qu'à initialiser ces données (les variables  $x$  et  $y$ , et la valeur  $c$ ) et ajouter la ligne `this.hook = new PalmConstraintPlugin(this);` pour créer une entité dont le but est de stocker les informations relatives aux explications (les valeurs retirées d'un domaine à cause de cette contrainte par exemple).

```
public class PalmGreaterOrEqualXYC extends AbstractPalmBinIntConstraint
{
    protected final int cste;

    public PalmGreaterOrEqualXYC(IntVar v0, IntVar v1, int cste) {
        this.v0 = v0;
        this.v1 = v1;
        this.cste = cste;
        this.hook = new PalmConstraintPlugin(this);
    }

    public String toString() {
        return this.v0 + " >= " + this.v1 + " + " + this.cste;
    }
}
```

On en profite pour définir une méthode `toString` permettant d'afficher la contrainte de manière lisible.

## 2. Propagation sur la contrainte

Tout d'abord, nous n'avons pas besoin de comportement particulier pour la première propagation de la contrainte, nous ne définissons donc ici que la méthode `propagate()`.

De plus, le premier filtrage revient à une situation où les bornes sont modifiées (on peut considérer que l'on passe de  $[-\#, \#]$  à un intervalle plus petit). La méthode se réduit donc au code suivant :

```
public void propagate() {
    this.awakeOnInf(1);
}
```

## Tutoriel : Expliquer une contrainte

```
    this.awakeOnSup(0);  
}
```

C'est tout ce que nous avons à faire ici.

### 3. Propagation sur les variables

A priori, `awakeOnInf` et `awakeOnSup` réagissent sur des modifications de domaine de variables non énumérées tandis que `awakeOnRem` réagit sur des variables énumérées. Cependant, cela n'implique en rien que l'autre variable a le même type ! Il faut donc bien prévoir ces deux types de variables.

En effet, nous pourrions gérer les variables énumérées comme les variables sur les bornes. Cependant, s'il s'agit de variables énumérées, il est souhaitable d'avoir les explications les plus précises possibles étant donné que le nombre de valeurs doit être relativement faible. Les méthodes de filtrage auront donc deux comportements selon que les variables soient énumérées ou non :

- si la variable dont on filtre des valeurs est énumérée, les explications pour chaque retrait doivent être aussi précises que possible puisque de toute manière chaque valeur retirée doit être expliquée séparément,
- par contre, si la variable dont on filtre des valeurs n'est pas énumérée, il suffit d'expliquer la nouvelle borne (dans le cas contraire, on risque de stocker un nombre d'explications trop important).

Par exemple, considérons le réveil sur la modification de la borne inférieure d'une variable. On doit réagir s'il s'agit de la variable `y` si elle `yinf+c` est supérieur strictement à `xinf` (on veut `x#y+c`). Cela donne alors :

```
public void awakeOnInf(int idx) {  
    if ((idx == 1) && (v1.getInf() + this.cste > v0.getInf())) {  
        if (v0.hasEnumeratedDomain()) {  
            int[] values = ((PalmIntVar) v0).getAllValues();  
            choco.palm.explain.Explanation expl =  
                new  
choco.palm.explain.GenericExplanation(this.getProblem());  
            ((PalmConstraintPlugin) this.getPlugIn()).self_explain(expl);  
            int index = 0;  
            int min = ((PalmIntDomain) v1.getDomain()).getOriginalInf();  
            while (index < values.length &&  
                values[index] < v1.getInf() + this.cste) {  
                for (int i = min; i <= values[index] - this.cste; i++)  
                    ((PalmIntVar) this.v1).self_explain(PalmIntDomain.VAL, i,  
expl);  
                ((PalmIntVar) this.v0).removeVal(values[index],  
                    this.cIdx0, expl.copy());  
                min = values[index] + 1 - this.cste;  
                index++;  
            }  
        } else {  
            choco.palm.explain.Explanation expl =  
                new  
choco.palm.explain.GenericExplanation(this.getProblem());  
            ((PalmConstraintPlugin) this.getPlugIn()).self_explain(expl);
```

```
((PalmIntVar) this.v1).self_explain(PalmIntDomain.INF, expl);
((PalmIntVar) this.v0).updateInf(this.v1.getInf() + this.cste,
    this.cIdx0, expl);
    }
}
}
```

Étudions plus précisément ce code selon le type de la variable  $x$  :

- si  $x$  est énumérée, on veut une explication par valeur retirée, donc il est intéressant de fournir une explication la plus précise possible; dans ce cas, après avoir ajouté dans l'explication la contrainte actuelle (en utilisant la méthode `getPlugIn()`), on calcule pour chaque valeur  $v$  à retirer du domaine de  $x$  une explication; plus la valeur sera importante, plus l'explication contiendra de justification de retraits de valeurs de  $y$ ;
- par contre si  $x$  n'est pas énumérée (on ne stocke que les bornes), il ne faut pas calculer toutes les explications sinon on risque d'utiliser beaucoup trop de mémoire pour stocker toutes les explications; on peut donc directement utiliser l'explication de la borne inférieure de  $y$ .

On notera que l'explication de la plus grande des valeurs retirées de  $x$  sera la même dans les deux cas : ce n'est que pour les valeurs intermédiaires que le résultat est différent.

La méthode `awakeOnSup` sera définie de manière similaire : il faut juste parcourir les valeurs dans l'ordre décroissant dans le cas d'une variable énumérée.

Pour ce qui est de `awakeOnRem`, il ne s'agit que d'un cas particulier des méthodes précédentes. Si l'on retire une valeur de  $y$ , il faut vérifier la borne inférieure de  $x$ ; si on retenir une valeur de  $x$ , il faut vérifier la borne supérieure de  $y$ . Cela donne :

```
public void awakeOnRem(int idx, int v) {
    if (idx == 0) {
        this.awakeOnSup(0);
    } else {
        this.awakeOnInf(1);
    }
}
```

#### 4. Propagation de la restauration

Lorsque les explications sont utilisées (notamment dans un cadre dynamique ou l'algorithme `mac-dbt`), des valeurs sont restaurées dans les domaines des variables lorsqu'une contrainte est retirée. Il faut alors vérifier que cette valeur est consistante ou non.

##### Warning:

Rien ne garantit lors de l'appel de la fonction que la valeur ou borne restaurée n'a pas déjà été supprimée. Le but est juste de vérifier si l'on peut retirer la valeur, et non pas de filtrer à cause de cette apparition de la valeur !

Dans le cas de l'inégalité, cela revient à vérifier la borne concernée. Par exemple, si la borne inférieure de  $x$  a été restaurée, il faut vérifier qu'elle est consistante. Pour cela il

## Tutoriel : Expliquer une contrainte

suffit de considérer que la borne inférieure de  $y$  a changé. Ainsi, on obtient les méthodes suivantes :

```
public void awakeOnRestoreInf(int idx) {
    if (idx == 0) this.awakeOnInf(1);
}

public void awakeOnRestoreSup(int idx) {
    if (idx == 1) this.awakeOnSup(0);
}
```

Le même raisonnement peut être tenu pour la restauration de valeur et non de bornes :

```
public void awakeOnRestoreVal(int idx, int val) {
    if (idx == 1) {
        this.awakeOnSup(0);
    } else {
        this.awakeOnInf(1);
    }
}
```

Cette étape de propagation après restauration est importante car elle garantit de ne pas trouver trop de solutions : si trop de solutions sont trouvées, il est fort probable qu'une phase de propagation lors de la restauration est mal codée.

## 5. Quelques méthodes utiles

whyIsTrue, isEntailed...