TP1 (suite)

Etape 5 : Représentation de contraintes en intension

Comme vous avez pu le constater, il peut être très fastidieux de coder les contraintes en extension. Nous allons nous intéresser ici au codage en intension de deux sortes de contraintes très répandues : les contraintes d'*égalité* (toutes les variables doivent avoir la même valeur) et les contraintes de *différence* (toutes les variables doivent avoir des valeurs deux à deux différentes).

➔ Restructurer le code de votre classe Constraint en considérant une hiérarchie de classes :

- `Constraint` devient une classe abstraite qui contient les attributs et méthodes communes à toutes les contraintes (attributs `num`, `name` et `varTuple` ; constructeurs initialisant `name` et `varTuples` ; méthodes associées à ces attributs : `arity`, etc.)
- `ConstraintExt` hérite de `Constraint` et définit les contraintes en extension. Ce type de contrainte est défini par un ensemble de tuples (`valTuples`) et possède notamment une méthode permettant d'insérer un tuple de valeurs (`addTuple`).
- `ConstraintDif` et `ConstraintEq` héritent également de `Constraint` et définissent respectivement les contraintes de différence et d'égalité.

La classe `Constraint` possède une méthode abstraite permettant de tester si un tuple de valeurs viole la contrainte. Cette méthode est appelée par la méthode de test de consistance de la classe `Solver` (elle-même appelée dans l'algorithme de backtrack pour tester la consistance d'une solution partielle). Elle est implémentée dans chacune des sous-classes : dans `ConstraintExt`, cette méthode cherche si un tuple de `valTuples` correspond au tuple de valeurs donné ; dans `ConstraintDif`, elle vérifie que toutes les valeurs du tuple donné sont deux à deux différentes ; dans `ConstraintEq`, elle vérifie que toutes les valeurs du tuple donné sont identiques. Attention la méthode de test de consistance de la classe `Solver` doit au préalable sélectionner les contraintes pour lesquelles il faut tester la violation ou non par l'assignation courante (cf. définition de la violation d'une contrainte dans le cours).

La méthode `toString` est également répartie dans la hiérarchie : la classe `Constraint` produit la partie de la chaîne de caractères correspondant à ses attributs, et les sous-classes font de même. Pensez à indiquer le type de la contrainte (« eq », « dif » et « ext » par exemple).

Il faut également étendre le format du fichier texte : par exemple, on peut prévoir une ligne au début de chaque contrainte comportant « ext », « eq » ou « dif ». Il n'y a que dans le cas de « ext » que des tuples de valeurs sont à lire. Pensez à répartir les tâches entre les constructeurs.

Quelques rappels de Java :

- un attribut « `protected` » est accessible par les sous-classes.
- Lorsqu'une méthode `m(…)` est redéfinie dans une classe, on peut faire appel à la super-méthode (la version de la méthode dans la super-classe directe) par le mot clé `super`. Usage : `super.m(…) ;`

- un constructeur peut faire appel à un constructeur de sa super-classe directe par le mot-clé `super`. Usage : `super(paramètres d'appel)` ; cette instruction doit être la première du constructeur.
- un constructeur peut de la même façon faire appel à un constructeur de la même classe par le mot-clé `this`.

Vous pouvez aller plus loin en représentant toutes sortes de contraintes en intension (dans ce cas, il faut prévoir une classe mère de toutes les contraintes en intension), et en leur associant une expression arithmétique construite à partir de leurs variables (il existe plusieurs bibliothèques java permettant d'évaluer des expressions).

➔ Testez vos nouvelles classes sur un petit problème de coloration, puis représentez et résolvez le problème du zèbre (TD 1). Certains exercices du TD 2 se prêtent également bien au test.

Etape 6 : Optimisations algorithmiques

Implémentez une heuristique d'ordonnancement des variables.

Remplacez l'algorithme de backtrack de base par l'algorithme de forward checking.