HAI710I - TP Modélisation en CSP

Choco *est une bibliothèque java permettant de modéliser et résoudre des problèmes de satisfaction de contraintes (cf. https://choco-solver.org/). Nous vous fournissons un* ***projet Maven*** *(voir plus loin :* `tp-ia-choco`*) qui intègre Choco. La java doc de Choco est disponible à l'adresse suivante :*

https://javadoc.io/doc/org.choco-solver/choco-solver/latest/org.chocosolver.solver/module-summary.html.

Les indications ci-dessous ont été testées dans l'environnement de développement ***Eclipse****. Mais le projet Maven fourni peut aussi être importé dans* ***IntelliJ****.*

A. Configurer son environnement de travail avec Eclipse

1) Si Eclipse n'est pas déjà installé dans votre espace personnel, faire l'installation. Suivre les instructions données ici : https://moodle.umontpellier.fr/mod/page/view.php?id=131074 . Dans Download, choisir x86_64. Ensuite lors de l'installation, "Eclipse IDE for Java Developers" suffit (c'est la première option de la liste).
2) Lancer Eclipse (un répertoire de travail par défaut `workspace` est créé).
3) Télécharger depuis Moodle le fichier *TP-IA-choco.zip*
4) Le dézipper et placer le répertoire obtenu `tp-ia-choco` dans le répertoire `workspace` créé par Eclipse. Ce répertoire correspond à un projet Eclipse.
5) Ouvrir ce projet sous Eclipse : sélectionner le menu « `Files` → `Open Projects from File System` ». Puis dans "`import source`" sélectionner le répertoire `tp-ia-choco` et cliquer sur « `Finish` ».

B. Modéliser le problème du Zèbre en extension

Dans l'environnement Eclipse, vous placerez vos fichiers java dans le répertoire `tp-ia-choco/src/main/java/` (qui correspond au package par défaut de votre projet). Pour l'instant, vous y trouverez le fichier `zebreExtension.java` qui contient un début de modélisation du problème du Zèbre où les contraintes sont données en extension.

La classe principale de Choco est la classe `Model` qui représente un réseau de contraintes (variables, domaines et contraintes) muni de diverses méthodes de résolution et récupération des solutions. On crée un modèle (ou : réseau) en lui donnant simplement un nom.

```
Model model = new Model("Zebre");
```

On déclare ensuite les variables et leurs domaines. Différents types de variables existent en Choco. Consulter les différentes méthodes de l'interface `IVariableFactory` pour avoir un aperçu des possibilités : https://javadoc.io/doc/org.choco-solver/choco-solver/latest/org.chocosolver.solver/org/chocosolver/solver/variables/IVariableFactory.html.

Pour ce TP nous utiliserons des variables entières, des instances de la classe `IntVar`. Une variable de ce type est créée par un appel à la méthode `intVar` de `Model`, en précisant un nom pour la variable et un ensemble de valeurs pour son domaine. Cet ensemble de valeur peut être défini par un tableau d'entiers, ou s'il s'agit d'un intervalle, en précisant la plus petite et plus grande valeur.

```
IntVar blu = model.intVar("Blue",new int []{1,2,3,4,5});
IntVar gre = model.intVar("Green",1,5);
```

On déclare ensuite les contraintes : ici on souhaite utiliser des contraintes définies en extension. En Choco, les contraintes en extension ne sont proposées que pour des variables de type `IntVar`. La création d'une contrainte en extension se fait en 3 étapes :

1) Créer un tableau de tuples de valeurs entières du type `int [][]`

```
int [][] tEq = new int [][]{{1,1},{2,2},{3,3},{4,4},{5,5}};
// la taille des tuples correspond à l'arité des contraintes
// que l'on veut créer
```

2) Créer une instance de la classe `Tuples` en précisant si les tuples donnés correspondent aux tuples autorisés (comme pour les contraintes en extension vues en cours) ou aux tuples interdits. Ci-dessous, la première ligne définit l'instance `tuplesautorisés` (ce qui permettra de définir une contrainte d'égalité) et la deuxième ligne définit l'instance `tuplesinterdits` (ce qui permettra de définir une contrainte de différence, à partir de la même liste de tuples).

```
Tuples tuplesAutorises = new Tuples(tEq,true);
Tuples tuplesInterdits = new Tuples(tEq,false);
```

3) Enfin, créer la contrainte par la méthode `table` qui prend en paramètre un tableau de variables entières `IntVar []` définissant sa portée et une instance de `Tuples` définissant ses tuples (autorisés ou interdits). Ensuite, cette contrainte est rendue « active » (i.e. elle sera prise en compte dans la recherche d'une solution) via la méthode `post()`. Ci-dessous on crée et active une contrainte de différence portant sur les variables entières `blu` et `gre` (celles que nou avons nommé "`Blue`" et "`Green`").

```
model.table(new IntVar[]{blu,gre},tuplesInterdits).post();
```

On peut afficher le réseau de contraintes construit ainsi :

```
System.out.println(model);
```

La méthode `getSolver()` de la classe `Model` retourne une instance de la classe `Solver` qui gère la recherche de solutions : https://javadoc.io/doc/org.choco-solver/choco-solver/latest/org.chocosolver.solver/org/chocosolver/solver/Solver.html).

La classe `Solver` offre notamment la méthode `solve()` qui recherche une (nouvelle) solution et retourne vrai si une (nouvelle) solution a été trouvée; dans ce cas chaque variable du réseau (`model`) contient la valeur qui lui a été affectée dans cette solution. Le code ci-dessous recherche une solution et affiche le réseau (dans l'affichage, "Satisfaction" indique si une solution a été trouvée - auquel cas elle est stockée dans les variables) :

```
if(model.getSolver().solve()) System.out.println("Solution trouvée");
else System.out.println("Pas de solution");

System.out.println(model);
```

Si on veut n'afficher que la solution trouvée, on peut l'enregistrer dans une instance de `Solution` :

```
Solver solver = model.getSolver();
Solution sol = new Solution(model);// récupère la liste des variables
if(solver.solve())    // une solution a été trouvée
    {
       sol.record(); // enregistre les valeurs des variables
       System.out.println(sol);
    }
```

Pour calculer toutes les solutions, il suffit de relancer la méthode `solve()` tant qu'elle retourne vrai. La méthode `getSolutionCount()` retourne le nombre de solutions trouvées au moment où elle est appelée. On peut aussi utiliser la méthode `findAllSolutions`() qui retourne la liste de toutes les solutions :

```
java.util.List<Solution> solutions = solver.findAllSolutions();
for (Solution sol : solutions)
      System.out.println(sol);
```

Enfin, un appel à la méthode `printStatistics()` permet d'avoir différentes informations sur le nombre de solutions trouvées, les temps d'exécutions, et le nombre de nœuds explorés.

```
model.getSolver().printStatistics();
```

Travail à faire :

1) Lire, comprendre et exécuter ce programme (méthode `main`).
2) Compléter ce programme en ajoutant les contraintes en extension correspondant aux phrases 2 à 15 du problème.
3) Quelle est la première solution trouvée ? Est-elle correcte ?
4) Calculez toutes les solutions. Combien en trouvez-vous ? Sont-elles toutes correctes ?

C. Modéliser le problème du Zèbre en intension

Choco dispose de nombreux types de contraintes. Pour les contraintes sur des variables entières, consulter la documentation de l'interface IIntConstraintFactory : https://javadoc.io/doc/org.choco-solver/choco-solver/latest/org.chocosolver.solver/org/chocosolver/solver/constraints/IIntConstraintFactory.html.

Parmi elles, les contraintes arithmétiques `arithm` de la forme :

```
model.arithm(x, op, y).post();
// avec x IntVar, et y IntVar ou int
// et op un comparateur "<", "<=", ">", ">=", "=" ou "!="

model.arithm(x, op2, y, op, z).post();
// avec x et y des IntVar, z un int,
// op un comparateur "<", "<=", ">", ">=", "=" ou "!=",
// et op2 est soit "+" soit "-" soit "*"
```

La contrainte `times` de la forme :

```
model.times(x, y, z).post();
// avec x et y des IntVar, et z IntVar ou int,
// contrainte x * y = z
```

La contrainte `distance` de la forme :

```
model.distance(x, y, op, z).post();
// avec x et y des IntVar, et z un int,
// op un comparateur "<", ">", "=" ou "!="
// contrainte |x - y | op z
```

Et la contrainte globale `allDifferent` :

```
model.allDifferent(vars).post();
// avec vars un tableau de IntVar
// contrainte de différence entre toutes les variables
```

Travail à faire :

1) Créer une classe `zebreIntension` qui modélise le problème du zèbre en intension.
2) Vérifier que vous obtenez les mêmes solutions que dans la version en extension.

D. Modéliser le problème des n-reines

Le problème des n-reines consiste à positionner *n* reines sur un échiquier de *nxn* cases de telles sortes qu'elles ne se menacent pas 2 à 2. On rappelle qu'aux échecs une reine peut se déplacer d'autant de cases qu'elle veut sur sa ligne, sa colonne et ses diagonales. Par conséquent deux reines ne doivent pas partager une même ligne, colonne ou diagonale. On souhaite écrire une classe `nReines` qui modélise ce problème (le *n* étant un paramètre).

1) Définir les variables (et domaines). L'idée étant d'avoir une variable R_i par reine. La variable R_i représentant la *reine de la ligne i* et sa valeur représentant la *colonne* où elle sera positionnée. La méthode `intVarArray` permet de créer un tableau de `IntVar` ayant toutes le même domaine.

```
IntVar [] t = model.intVarArray("x",5,1,25);
// crée un tableau de 5 variables entières de domaine [1,25]
```

2) Quelles contraintes faut-il mettre entre les variables R_i pour interdire qu'elles soient sur la même ligne ?
3) Quelles contraintes faut-il mettre entre les variables R_i pour interdire qu'elles soient sur la même colonne ?
4) Soit R_i la colonne de la reine sur la ligne i et R_j la colonne de la reine sur la ligne j. Quelle contrainte arithmétique entre R_i et R_j permet d'interdire que ces deux reines soient sur la même diagonale ?
5) Quelles contraintes faut-il mettre entre les variables R_i pour interdire qu'elles soient sur la même diagonale ?
6) Rechercher une solution au problème des n-reines pour *n=1, 2, 3, 4, 8, 12, 16* ? Que remarquez-vous ?
7) Combien de solutions y-a-t-il à ce problème pour ces différents *n* ?