TP CSP - Modélisation

Choco est une librairie java permettant de modéliser et résoudre des problèmes de satisfaction de contraintes (cf. https://choco-solver.org/). Nous vous proposons de réaliser le TP sous Eclipse en accédant à la librairie via le gestionnaire de modules Maven. La javadoc de la librairie Choco est disponible à l'adresse : https://javadoc.io/doc/org.choco-solver/choco-solver/latest/org.chocosolver.solver/module-summary.html.

A. Configurer son environnement de travail

1) Installer Eclipse sur votre poste de travail : Cf. https://moodle.umontpellier.fr/mod/page/view.php?id=131074
2) Lancer Eclipse (un répertoire de travail par défaut `workspace` est créé)
3) Télécharger depuis Moodle le fichier *TP-IA-choco.zip*
4) Dézipper-le et placer le répertoire `TP-IA-choco` à l'endroit souhaité (a priori dans le répertoire `workspace` créé par Eclipse).
5) Dans Eclipse, sélectionner menu « `Files` → `Open Projects from File System` »
6) Sélectionner le répertoire `TP-IA-choco` puis « `Finish` »

B. Modéliser le problème du Zèbre en extension

Le fichier `zebreExtension.java` 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 permet l'accès à la fois au réseau de contraintes (variables, domaines et contraintes) et aux diverses méthodes de résolution et de récupération des solutions. On crée simplement un modèle en lui donnant un nom.

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

On déclare ensuite les variables et leur 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 polymorphe `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 `IntVar`. La création d'une contrainte en extension se fait en :

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

```
int [][] tEq = new int [][]
{{1,1},{2,2},{3,3},{4,4},{5,5}};
```

2) Créant un objet de la classe `Tuples` qui permet de préciser si les tuples donnés définiront la contrainte en extension en précisant ses tuples autorisés (cf. exemple tuplesAutorisés), ou ses tuples interdits (cf. exemple tuplesInterdits).

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

3) Créant 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). Puis rendant cette contrainte « active » (i.e. elle sera prise en compte dans la recherche d'une solution) via la méthode `post()`.

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

On peut afficher le réseau de contraintes saisi par un simple affichage :

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

La méthode `getSolver()` donne accès aux mécanismes de résolutions de Choco (cf. méthodes de la classe Solver https://javadoc.io/doc/org.choco-solver/choco-solver/latest/org.chocosolver.solver/org/chocosolver/solver/Solver.html). On peut vérifier l'existence d'une solution au réseau de contraintes construit par la méthode `solve()` qui retourne vrai si une solution a été trouvée. On peut alors afficher cette solution par un simple affichage du modèle qui donne à tout moment de la résolution l'assignation courante. Pour calculer toutes les solutions, il suffit de relancer la méthode `solve()` tant qu'elle renvoie vrai.

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

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

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.
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 un IntVar et y soit un IntVar soit un int
// et op un comparateur ">=", ">", "<=", "<", "=" ou "!="

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

La contrainte valeur absolue `absolute` de la forme :

```
model.absolute(x,y).post();
// avec x et y IntVar (x = |y|)
```

Et la contrainte globale `allDifferent` :

```
model.allDifferent(vars).post();
// avec vars un tableau de IntVar
```

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 *n*x*n* 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 `nReine` 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* ?