UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F-302 : Informatique Fondamentale

Problèmes de Satisfactions de Contraintes et Utilisation de l'Outil ChocoSolver

Rapport

Auteurs :

Hakim Boulahya

Youcef Bouharaoua

Professeur :

Emmanuel Filiot

21 mai 2017

Table des matières

1 Problèmes d'échecs (Question 1, 2 et 3)

Remarque Pour les deux premières questions *i.e.* le problème d'indépendance et le problème de domination, nous utilsons le même ensemble de variables, seul une partie des contraintes diffèrent.

1.1 Variables et notations du CSP

Notons $G(n, k1, k2, k3)$ une instance de notre problème. Chaque case du jeu est représenté par une valeur $i \in I$ où $I = \{1, 2, ..., n^2\}$. Chaque type de pièce est réprésenté par une variable $b \in B$ où $B = \{t(our), f(ou), c(avalier)\}$. Le nombre de pièce de chaque type est représenté par k_b où $k_t = k1$, $k_f = k2$ et $k_c = k3$. Pour chaque type de pièce $b \in B$ faisant partie du problème nous utilisons un ensemble de variables booléennes par case. L'ensemble de variables $X : \{x_{bi} | \forall b \in B, \forall i \in I\}$. La variable x_{bi} est vrai si une pièce de type b se situe sur la case i, faux sinon. Nous notons l'ensemble des variables $P(resence)$ où $P = \{p_i | \forall i \in I\}$, qui est vrai lorsqu'une pièce de n'importe quelle type se trouve sur la case i.

1.2 Fonction d'attaque (Question Bonus)

1.2.1 Définition

La fonction d'attaque d'une pièce de type $b \in B$ est une fonction $f_b(i, j)$ $i, j \in I$ qui retourne vrai si une pièce de type b sur la case i peut attaquer une pièce (quelconque) sur la case j. Nous avons donc trois fonctions, où ir et jr correspondent aux lignes des cases respectives, et ic et jc aux colonnes :

- $f_t(i, j) := ir = jr \vee ic = jc$
- $f_f(i, j) := |ir - jr| = |ic - jc|$
- $f_c(i, j) := (|ir - jr| = 2 \wedge |ic - jc| = 1) \vee (|ir - jr| = 1 \wedge |ic - jc| = 2)$

1.2.2 Implémentation

Pour implémenter cette fonction, nous utilisons des tableaux de la forme `pieceAttacks[i][j]`. Par exemple la fonction $f_t(i, j)$ est représenté par le tableau `towerAttacks[i][j]`, créer à partir de la classe statique `AttackFactory`.
Cette classe comporte les méthodes `AttackFactory.towerAttacks(n, m)`, `AttackFactory.foolAttacks(n, m)` et `AttackFactory.knightAttacks(n, m)`. Ces méthodes retournent les tableaux respectivent représentant la fonction $f_b(i, j)$.

1.3 Contraintes

1.3.1 Nombre de pièce

Le nombre de pièce de chaque type doit être égal aux valeurs de l'instance du problème :

$$\sum_{i \in I} x_{bi} = k_b \qquad \forall b \in B \tag{1}$$

Une somme est représenter par la méthode `sum` du modèle, `vars` représenté le tableau x_b et k représente la valeur k_b. Cette contrainte est créée pour toutes les pièces i.e. $\forall b \in B$, il y a donc trois contraintes dans notre cas précis.

```
model.sum(vars, "=", k).post();
```

1.3.2 Contrainte de présence

Pour toutes cases, la variable de présence doit être vrai si une pièce s'y trouve :

$$p_i = \bigvee_{b \in B} x_{bi} \qquad \forall i \in I \tag{2}$$

Cette contrainte est modéliser en utilisant la méthode `arithm` du modèle :

```
model.arithm(presenceVars[i], "=", model.or(towerVars[i], foolVars[i], knightVars[i]).
reify()).post();
```

1.3.3 Un élément par case

Une seule pièce peut se trouver sur une case i :

$$x_{bi} \rightarrow \neg x_{b'i} \equiv \neg x_{bi} \vee \neg x_{b'i} \qquad \forall i \in I, \quad \forall b, b' \in B | b \neq b' \tag{3}$$

Pour chaque case, if faut créer 3 contraintes de la forme (`var1` représente x_{bi} et `var2` $x_{b'i}$) :

```
model.or(model.arithm(var1, "=", 0), model.arithm(var2, "=", 0)).post();
```

1.4 Problème d'indépendance : aucune pièce n'est vulnérable (Question 1)

Pour toute pièce, celle ci ne doit être menacé par aucune autre pièce :

$$x_{bi} \rightarrow (p_j \rightarrow \neg f_b(i,j)) \equiv \neg x_{bi} \vee \neg p_j \vee \neg f_b(i,j) \qquad \forall i, j \in I | i \neq j, \quad \forall b \in B \tag{4}$$

Implémentation de la contrainte :

```
private void postIndependenceConstraints() {
    postIndependenceConstraint(towerVars, towerAttacks);
    postIndependenceConstraint(foolVars, foolAttacks);
    postIndependenceConstraint(knightVars, knightAttacks);
}

private void postIndependenceConstraint(BoolVar[] pieceVars, boolean[][] pieceAttacks) {
    for (int i = 0; i < getNumberOfElements(); i++) {
        for (int j = 0; j < getNumberOfElements(); j++) {
            if (i != j) {
                model.or(model.arithm(pieceVars[i], "=", 0),
                        model.arithm(presenceVars[j], "=", 0),
                        model.arithm(model.boolVar(pieceAttacks[i][j]), "=", 0)).post();
            }
        }
    }
}
```

1.5 Problème de domination : toutes les pièces sont vulnérables (Question 2)

$$\bigvee_{j \in I | j \neq i} \neg p_i \bigvee_{b \in B} (x_{bj} \wedge f_b(j,i)) \qquad \forall i \in I \tag{5}$$

Implémentation de la contrainte :

```
1
2  private void postDominationConstraints() {
3      for (int i = 0; i < getNumberOfElements(); i++) {
4          List<Constraint> constraintList = new ArrayList<>();
5          for (int j = 0; j < getNumberOfElements(); j++) {
6              if (i != j) {
7                  Constraint dominationPerJ = model.or(model.arithm(presenceVars[i], "=", 0),
8                          buildAttackedByConstraint(model, towerVars, towerAttacks, i, j),
9                          buildAttackedByConstraint(model, towerVars, towerAttacks, i, j),
10                         buildAttackedByConstraint(model, knightVars, knightAttacks, i, j));
11                 constraintList.add(dominationPerJ);
12             }
13         }
14         Constraint[] constraints = new Constraint[constraintList.size()];
15         constraintList.toArray(constraints);
16         model.or(constraints).post();
17     }
18 }
19
20 public static Constraint buildAttackedByConstraint(Model model, BoolVar[] pieceVars, boolean
       [][][] pieceAttacks, int i, int j) {
21     return model.and(model.arithm(pieceVars[j], "=", 1),
22             model.arithm(model.boolVar(pieceAttacks[j][i]), "=", 1));
23 }
```

2 Domination d'un échiquier par des cavaliers (Question 4)

2.1 Variables et notations

Pour modéliser ce problème nous allons utiliser les variables suivantes (défini précédemment) : x_{ci}, $f_{i,j}$ et une nouvelle variable *totalCavalier* qui correspond au nombre de cavalier placé sur l'échiquier. Le domaine de *totalCavalier* est $D = \{1, 2, ..., n^2\}$.

2.2 Contraintes

2.2.1 Nombre de cavalier

$$totalCavalier = \sum_{i \in I} x_{ci} \tag{6}$$

2.2.2 Domination de toutes les cases

Cette contrainte suit le même principe que la contrainte (5) du problème de domination, avec comme différence le faite que ce sont toutes les cases qui doivent être dominées et non pas seulement les cases où se

situent des pièces :

$$\bigvee_{j \in I | j \neq i} x_{ci} \vee (x_{bj} \wedge f_b(j,i)) \qquad \forall i \in I \tag{7}$$

2.3 Minimisation du nombre de cavalier

Pour minimiser le nombre de cavalier, il suffit d'indiquer que l'objectif du programme est la minimisation de la variable *totalCavalier* :

```
model.sum(knightVars, "=", totalOfKnights).post();
model.setObjective(Model.MINIMIZE, totalOfKnights);
```

3 Surveillance du musée

3.1 Variables et notations du problèmes

Notons $G(map, n, m)$ l'instance du problème où *map* représente la carte avec les obstacles, n le nombre de ligne et m le nombre de colonnes de la carte. Nous avons l'ensemble $I = \{1, 2, ..., n \times m\}$ qui représente l'ensemble des cases sur la carte (les cases libres et les cases avec obstacles). Notons l'ensemble $E = \{i \in I | i$ est une case vide i.e. sans obstacle$\}$. Nos variables sont les suivantes : l'ensemble de variables $L = \{l_i | \forall i \in I\}$ où l_i est une variable booléenne qui est vrai si un laser se trouve sur la case i, faux sinon. L'ensemble des variables $W = \{w_e | \forall e \in E\}$ où le domaine de w_e est $D_{w_e} = \{$ensemble des cases nord, sud, est, ouest accessibles par e$\}$. Le domaine D_{w_e} correspond donc à l'ensemble des cases accessibles, suivant l'un des 4 points cardinaux et sans obstacles entre eux, par e. La variable w_e indique la case où se trouve le laser surveillant la case e. L'ensemble de variable $D = \{d_e | \forall e \in E\}$, où le domaine de d_e est $D_{d_e} = \{NORD, SUD, EST, WEST, SELF\}$. La valeur de la variable d_e indique la direction où se trouve le surveillant de la case e, soit w_e. Si la case e est un laser il n'est pas nécessaire de la surveiller, et donc la valeur de d_e correspond à $SELF$. Pour permettre de minimiser le nombre de lasers, nous utilisons une variables *totalLaser* dont le domaine est $\{1, 2, ..., n \times m\}$.

3.2 Fonction de direction

Pour respecter la condition d'un laser stipulant qu'il ne peut uniquement surveiller une direction, nous allons créer une fonction que nous utilisons dans les contraintes. La fonction $g(e, i)$ où $e \in E$ et $i \in D_{w_e}$. Cette fonction renvoie une valeur $d \in D_{d_e}$ indiquant la direction dans laquelle la case i se trouve par rapport a e. Cette fonction va nous permettre de verifier qu'un laser surveille des éléments dans une et une seule direction, et uniquement des cases dans sont champs de vision *i.e.* accessible sans obstacle.

3.3 Contraintes

3.3.1 Nombre de laser

$$totalLaser = \sum_{i \in I} l_i \tag{8}$$

3.3.2 Identification des lasers

Tous les surveillants des cases libres doivent être des lasers :

$$l_{w_e} = 1 \qquad \forall e \in E \tag{9}$$

Cette contrainte est implémentée en utilisant la méthode `element` du modèle :

```
model.element(model.intVar(1), laserVars, watcherVars.get(i)).post();
```

3.3.3 Direction des surveillants

La direction d'une case doit être égal à la direction vers le surveillant :

$$d_e = g(e, w_e) \qquad \forall e \in E \tag{10}$$

Cette contrainte est également implémenté en utilisant la méthode `element`, car la fonction $g(e, i)$ est réprésenté par un tableau :

```
model.element(dirOfVars.get(i), getDirectionInTable(i), watcherVars.get(i)).post();
```

3.3.4 Un laser ne vise qu'une seule direction

Si deux éléments ne sont pas des lasers, et que le surveillant est le même, la direction doit être également la même :

$$(d_i \neq SELF) \wedge (d_j \neq SELF) \rightarrow ((w_i = w_j) \rightarrow (d_i = d_j)) \qquad \forall i, j \in E \tag{11}$$

En simplifiant les implications, cela donne :

$$(d_i = SELF) \vee (d_j = SELF) \vee (w_i \neq w_j) \vee (d_i = d_j) \qquad \forall i, j \in E \tag{12}$$

Implémentation de la contrainte :

```
private void postDirectionConstraints() {
    postDirectionOfConstraints();
    for(Integer i : getEmptyElements()){
        for(Integer j : getEmptyElements()) {
            model.or(buildAreLasers(i, j),
                     buildDifferentWatcher(i, j),
                     buildSameDirection(i, j)).post();
        }
    }
}

private Constraint buildAreLasers(Integer i, Integer j){
    return model.or(model.arithm(dirOfVars.get(i), "=", SELF),
                    model.arithm(dirOfVars.get(j), "=", SELF));
}

private Constraint buildDifferentWatcher(Integer i, Integer j){
    return model.arithm(watcherVars.get(i), "!=", watcherVars.get(j));
}
```

```
20
21 private Constraint buildSameDirection(Integer i, Integer j) {
22     return model.arithm(dirOfVars.get(i), "=", dirOfVars.get(j));
23 }
```

3.4 Minimisation du nombre de laser

Pour minimiser le nombre de laser, il suffit d'indiquer que l'objectif du programme est la minimisation de la variable *totalLaser* :

```
1 model.sum(laserVars, "=", numberOfLasersVar).post();
2 model.setObjective(Model.MINIMIZE, numberOfLasersVar);
```

4 Utilisation des outils

Trois outils ont été implémentés (sous forme de classes Java) : `Chess(n, k1, k2, k3, domination)`, `MuseumSurveillance(filename)` et `KnightFullDomination(n)`.

Remarque Tous les outils ont été testés sous OpenJDK8.

Tous les fichiers **jar** permettant d'exécuter les outils se trouvent dans le dossier **tools**.

Pour exécuter le programme `ChessSolver`, placé vous à la source et exécuté :

```
java -jar tools/chess.jar -d 1 -n 3 -t 2 -f 1 -c 1
```

Pour exécuter `MuseumSurveillance`, par defaut le programme utilise le fichier `input/museum.txt` :

```
java -jar tools/museum.jar <filename>
```

Pour exécuter `KnightFullDomination` :

```
java -jar tools/knight.jar 4
```

Tous les fichiers sources du projet se trouve dans le dossier **src**.