Projet : INFO-F-302 Informatique Fondamentale.

George Rusu et Maximilien Romain

21 mai 2017

Table des matières

1 Introduction

Le premier objectif de ce projet est de modeliser divers problemes en problemes de satisfaction de contraintes (CSP). Le second objectif est d'implementer un programme resolvant ces problemes en utilisant ChocoSolver.

link : http ://www-master.ufr-info-p6.jussieu.fr/2005/IMG/pdf/rp3.pdf

2 Question 1

L'ensemble des cases du jeux V où $\#V = n^2$
Variables de décision $X = \{x_{i,j} | \forall i, j (1 \leq i \leq n).(1 \leq j \leq n)\}$, n^2 variables de décision
Domaines : $D = \{vide, fous, cavalier, tour\}$
Contraintes : Pour chacune des pièces de l'echequier, si une case est occupé par un pions, alors dans la porté de ce pion, donc les cases attaquables par ce pion, doivent être obligatoirement vide. On parcours donc chaque case, et lorsque qu'une case est occupé, en fonction du pion qui occupe cette case, les contraintes changent.

2.1 Contraintes tour

Pour chaque colonne de l'echequier, si une case de cette colonne est occupée par une tour, alors le reste des cases de cette colonne doivent être vide :

$$c_{T_{col},j} = ((x_{1,j}, x_{2,j}, \ldots, x_{n,j}), \{(b_1, b_2, \ldots, b_n) | b_i = T, b_j = V, \forall j \neq i\})$$

Pour chaque ligne de l'echequier, si une case de cette ligne est occupée par une tour, alors le reste des cases de cette ligne doivent être vide :

$$c_{T_{i,ligne}} = ((x_{i,1}, x_{i,2}, \ldots, x_{i,n}), \{(b_1, b_2, \ldots, b_n) | b_i = T, b_j = V, \forall j \neq i\})$$

2.2 Contraintes fou

Les contraintes du fou sont composées dans le même état d'esprit que les contraintes des tours, cepandant la portée d'un fou change, et en fonction de la digonale sur laquelle est placé le fou, la distance maximal que ce fou peut atteindre change. En effet les plus grand diagonales se trouve en $(0, 0)$ et $(0, n-1)$. Par contrainte, on considère toujours la diagonale opposée, et on réunie les deux diagonales par un *and*.

$$c_{F,2*n-2} = ((x_{1,n-1}, x_{2,n}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\}) \wedge$$
$$((x_{1,2}, x_{2,1}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\})$$

$$\vdots$$

$$c_{F,n+1} = ((x_{1,2}, x_{2,3}, \ldots, x_{n-1,n}), \{(b_1, b_2, \ldots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\}) \wedge$$
$$((x_{1,n-1}, x_{2,n-2}, \ldots, x_{n-1,1}), \{(b_1, b_2, \ldots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\})$$
$$c_{F,n} = ((x_{1,1}, x_{2,2}, \ldots, x_{n,n}), \{(b_1, b_2, \ldots, b_n) | b_i = F, b_j = V, \forall j \neq i\}) \wedge$$
$$((x_{1,n}, x_{2,n-1}, \ldots, x_{n,1}), \{(b_1, b_2, \ldots, b_n) | b_i = F, b_j = V, \forall j \neq i\})$$
$$c_{F,n-1} = ((x_{2,1}, x_{3,2}, \ldots, x_{n,n-1}), \{(b_1, b_2, \ldots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\}) \wedge$$
$$((x_{2,n}, x_{3,n-1}, \ldots, x_{n,2}), \{(b_1, b_2, \ldots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\})$$

$$\vdots$$

$$c_{F,2} = ((x_{n-1,1}, x_{n,2}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\}) \wedge$$
$$((x_{n-1,n}, x_{n,n-1}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\})$$

Les diagonales de gauche à droite sont représenté par la matrice ci-dessous.

$$\begin{pmatrix} c_{F,n} & c_{F,n+1} & \cdots & c_{F,2*n-2} & \\ c_{F,n-1} & c_{F,n} & c_{F,n+1} & \vdots & c_{F,2*n-2} \\ \vdots & c_{F,n-1} & c_{F,n} & c_{F,n+1} & \vdots \\ c_{F,2} & \vdots & c_{F,n-1} & c_{F,n} & c_{F,n+1} \\ & c_{F,2} & \cdots & c_{F,n-1} & c_{F,n} \end{pmatrix} \quad (1)$$

2.3 Contraintes cavalier :

Les contraintes des cavaliers sont plus simple. En effet si on rencontre une case occupée par un cavalier, alors il n'y a que 8 contraintes pour les 8 cases que le cavalier peut atteindre. Si le cavalier est en (i, j), alors il ne peut qu'attaquer les cases où $i + e, j + e, e \in [+1, -1, +2, -2]$

$$c_{C,(i,j)} = ((x_{i,j}, x_{i+1,j+2}, x_{i+1,j-2}, x_{i-1,j+2}, x_{i-1,j-2}, x_{i+2,j+1}, x_{i+2,j-1}, x_{i-2,j+1}, x_{i-2,j-1}),$$
$$\{(b_1, b_2, \ldots, b_9)|b_1 = C, b_2, \ldots b_9 = V\})$$

3 Question 2

L'ensemble des cases du jeux V où $\#V = n^2$
Variables de décision $X = \{x_{i,j}|\forall i, j(1 \leq i \leq n).(1 \leq j \leq n)\}$, n^2 variables de décision
Domaines : $D = \{Vide, tour, fous, cavalier\}$
Contraintes : Pour chacune des pièces de l'echequier, si une case est vide, alors il existe au moins un pion qui menace cette case vide

3.1 Contraintes tours

Pour chaque case de l'echequier qui n'est pas occupé par un pion, alors il existe au moins une tour dans la ligne ou la colonne de la case non occupée.

$$c_T = ((x_{i,j}, \forall i, j(1 \leq i \leq n).(1 \leq j \leq n)),$$
$$\{b_{i,j}, \forall i, j(1 \leq i \leq n).(1 \leq j \leq n)|b_{i,j} = V, \forall l, (1 \leq l \leq n)\exists b_{l,j} = T \vee b_{i,j} = V, \forall k(1 \leq k \leq n), \exists b_{i,k} = T\})$$

Nous devons aussi indiqué qu'il n'y pas d'autre pion entre la case vide et la tour. On indique donc que les cases entre la case vide et la case occupé par une tour, doivent non occupées. Nous avons donc une telle contrainte pour la ligne et la colonne de la case vide menacé par une tour.

$$c_{couple,T_{colonnes}} = ((x_{i,j}, x_{l,j}), \{(b_{i,j}, b_{l,j})|b_{i,j} = T, b_{l,j} = V, \rightarrow b_{m,j}\forall m(i < m < l), b_{m,j} = V\})$$
$$c_{couple,T_{lignes}} = ((x_{i,j}, x_{i,k}), \{(b_{i,j}, b_{i,k})|b_{i,j} = T, b_{i,k} = V, \rightarrow b_{i,m}\forall m(j < m < k), b_{i,m} = V\})$$

3.2 Contraintes fous

Pour chaque case de l'echequier qui n'est pas occupé par un pion, alors il existe au moins un fou dans les deux diagonales de la case vide.

$$c_F = ((x_{i,j}, \forall i, j(1 \leq i \leq n).(1 \leq j \leq n)),$$
$$\{b_{i,j}, \forall i, j(1 \leq i \leq n).(1 \leq j \leq n)|b_{i,j} = V, \forall l, k(1 \leq l \leq n).(1 \leq k \leq n)\exists b_{l,k} = F \vee$$
$$b_{i,j} = V, \forall l, k(1 \leq l \leq n).(n \leq k \leq 1), \exists b_{l,k} = F\})$$

De nouveau nous devons nous assurer qu'il n'y a pas d'autre pions qui gène le fou entre la case vide et ce fou.

$$c_{couple,F_{diag1}} = ((x_{i,j}, x_{l,k}), \{(b_{i,j}, b_{l,k})|b_{i,j} = T, b_{l,k} = V, \rightarrow b_{x,y}\forall x, y(i < x < l).(j < y < k), b_{x,y} = V\})$$
$$c_{couple,F_{diag2}} = ((x_{i,j}, x_{l,k}), \{(b_{i,j}, b_{l,k})|b_{i,j} = T, b_{l,k} = V, \rightarrow b_{x,y}\forall x, y(i < x < l).(k < y < j), b_{x,y} = V\})$$

3.3 Contraintes cavaliers :

Pour chaque case de l'echequier qui n'est pas occupé par un pion, alors il existe au moins un cavalier qui menace la case vide. Pour cela il suffit de verifier qu'il existe une case dans les 8 cases correspondant à l'ensemble des mouvements du cavalier ($\{+1, -1, +2, -2\}$), soit occupé par un cavalier.

$$c_{C,(i,j)} = ((x_{i,j}, x_{i+1,j+2}, x_{i+1,j-2}, x_{i-1,j+2}, x_{i-1,j-2}, x_{i+2,j+1}, x_{i+2,j-1}, x_{i-2,j+1}, x_{i-2,j-1}),$$
$$\{(b_1, b_2, \ldots, b_9) | b_1 = V, \forall i, j \in [+1, +2, -1, -2], \exists b_{i,j} = C\})$$

3.4 Contrainte finale :

Finalement il n'y a qu'une seule grande contrainte à appliquer à notre problème. Etant donné qu'il suffit qu'un seule pion menace une case vide, nous pouvons relier nos contrainte par des conditions *or*.

$$C = (C_T \wedge c_{couple,T_{colonnes}} \wedge c_{couple,T_{lignes}}) \vee (c_F \wedge c_{couple,F_{diag1}} \wedge c_{couple,F_{diag2}}) \vee c_{C,(i,j)}$$

4 Question 2 avec ensemble

L'ensemble des cases du jeux V où $\#V = n^2$
Variables de décision $X = \{x_{i,j} | \forall i, j (1 \leq i \leq n).(1 \leq j \leq n)\}$, n^2 variables de décision
Domaines : $D = \{Vide, tour, fous, cavalier\}$
Contraintes : Pour chacune des pièces de l'echequier, si une case est vide, alors il existe au moins un pion qui menace cette case vide. Le problème ici, est que nous devons trouver un moyen d'appliquer un *OR* de contraintes, ce qui n'est pas possible. Nous allons donc définir divers ensemble, et ensuite écrire une grande contrainte appliquant des *Unions* de ces ensembles.

4.1 Ensemble Tours

Pour chaque case de l'echequier qui n'est pas occupé par un pion, alors il existe au moins une tour dans la ligne ou la colonne de la case non occupée.

$$e_T = \{(x_{i,j}, \forall i, j (1 \leq i \leq n).(1 \leq j \leq n)),$$
$$\{b_{i,j}, \forall i, j (1 \leq i \leq n).(1 \leq j \leq n) | b_{i,j} = V, \forall l, (1 \leq l \leq n) \exists b_{l,j} = T \vee b_{i,j} = V, \forall k (1 \leq k \leq n), \exists b_{i,k} = T\}\}$$

Nous devons aussi indiqué qu'il n'y pas d'autre pion entre la case vide et la tour. On indique donc que les cases entre la case vide et la case occupé par une tour, doivent non occupées. Nous avons donc une telle contrainte pour la ligne et la colonne de la case vide menacé par une tour.

$$e_{couple,T_{colonnes}} = \{(x_{i,j}, x_{l,j}), \{(b_{i,j}, b_{l,j}) | (b_{i,j} = T, b_{l,j} = V) \wedge (\forall m (i < m < l), b_{m,j} = V)\}\}$$
$$e_{couple,T_{lignes}} = \{(x_{i,j}, x_{i,k}), \{(b_{i,j}, b_{i,k}) | (b_{i,j} = T, b_{i,k} = V) \wedge (\forall m (j < m < k), b_{i,m} = V)\}\}$$

4.2 Ensemble fous

Pour chaque case de l'echequier qui n'est pas occupé par un pion, alors il existe au moins un fou dans les deux diagonales de la case vide.

$$e_F = \{(x_{i,j}, \forall i, j (1 \leq i \leq n).(1 \leq j \leq n)),$$
$$\{b_{i,j}, \forall i, j (1 \leq i \leq n).(1 \leq j \leq n) | b_{i,j} = V, \forall l, k (1 \leq l \leq n).(1 \leq k \leq n) \exists b_{l,k} = F \vee$$
$$b_{i,j} = V, \forall l, k (1 \leq l \leq n).(n \leq k \leq 1), \exists b_{l,k} = F\}\}$$

De nouveau nous devons nous assurer qu'il n'y a pas d'autre pions qui gène le fou entre la case vide et ce fou.

$$e_{couple,F_{diag1}} = \{(x_{i,j}, x_{l,k}), \{(b_{i,j}, b_{l,k}) | (b_{i,j} = T, b_{l,k} = V) \wedge (\forall x, y (i < x < l).(j < y < k), b_{x,y} = V)\}\}$$
$$e_{couple,F_{diag2}} = \{(x_{i,j}, x_{l,k}), \{(b_{i,j}, b_{l,k}) | (b_{i,j} = T, b_{l,k} = V) \wedge (\forall x, y (i < x < l).(k < y < j), b_{x,y} = V)\}\}$$

4.3 Ensemble cavaliers :

Pour chaque case de l'échiquier qui n'est pas occupé par un pion, alors il existe au moins un cavalier qui menace la case vide. Pour cela il suffit de verifier qu'il existe une case dans les 8 cases correspondant à l'ensemble des mouvements du cavalier ($\{+1, -1, +2, -2\}$), soit occupé par un cavalier.

$$e_{C,(i,j)} = \{(x_{i,j}, x_{i+1,j+2}, x_{i+1,j-2}, x_{i-1,j+2}, x_{i-1,j-2}, x_{i+2,j+1}, x_{i+2,j-1}, x_{i-2,j+1}, x_{i-2,j-1}),$$
$$\{(b_1, b_2, \ldots, b_9) | b_1 = V, \forall i, j \in \{+1, +2, -1, -2\}, \exists b_{i,j} = C\}\}$$

4.4 Contrainte finale :

Finalement il n'y a qu'une seule grande contrainte à appliquer à notre problème. Etant donné qu'il suffit qu'un seule pion menace une case vide, nous pouvons relier nos ensemble par des *unions* $\cup$ et des *intersections* de contraintes $\cap$.

$$C = (e_T \cap e_{couple,T_{colonnes}} \cap e_{couple,T_{lignes}}) \cup (e_F \cap e_{couple,F_{diag1}} \cap e_{couple,F_{diag2}}) \cup e_{C,(i,j)}$$

5 Question 3

Pour cette question nous avons du implementer les deux problèmes décrit ci-dessus. Nous sommes tout d'abord parti sur le fait de faire tout dans une fonction. Nous nous sommes vites rendu compte que cela n'était pas du tout visible est très difficile à debugger. Après de nombreuse recherche et de nombreuse question posé à l'assistant, nous avons eu le déclic que le problème de domination est l'inverse de l'indépendance. En effet, si une piece menace une autre, il s'agit du problème de domination et inversement pour le problème d'indépendance, une piece ne doit pas menacer une autre. La negation fait donc son apparition.

Pour pouvoir utiliser ce concept avec la negation, nous avions du changer notre premier mode de representation à savoir celui avec un échiquier de taille $n * n$ où chaque case peut-être soit une Tour soit un Cavalier soit un Fou soit un Vide. Nous nous sommes orienté vers la programmation Orienté Objet. Nous avons decider que puisque on va manipuler des pieces, on pourrait créer une classe Piece qui représenterai une piece du Jeu. Mais chaque type de piece a son propre espace de jeu (ses propres contraintes sur les mouvements), nous avons donc du faire de cette classe Piece une classe abstraite afin de pouvoir pour chaque type redéfinir son espace de jeu.

5.1 Lancement du programme

Lors du lancement du jeu, il faut choisir les paramètres en ligne de commande de la manière suivante :

```
Georges-MacBook-Pro:dist georgerusu$ java -jar q3.jar -i -n 4 -t 2 -f 1 -c 2
Probleme d'indepandance
* * F C
* * C *
* T * *
T * * *
```

FIGURE 1 – Exemple de lancement problème independance

où

- -i si c'est le problème d'indépendance
- -d si c'est le problème de domination
- -n 4 pour la taille de l'échiquier (ici nous avons défini une taille de 4)
- -t 2 pour le nombre de piece Tour (ici 2)
- -f 1 pour le nombre de piece Fou (ici 1)
- -c 2 pour le nombre de Cavalier (ici 2)

Encore un exemple d'utilisation, cette fois-ci pour le problème de domination :

```
Georges-MacBook-Pro:dist georgerusu$ java -jar q3.jar -d -n 4 -t 2 -f 1 -c 2
Probleme de domination
* C * *
* C * F
T * * *
* * T *
```

FIGURE 2 – Exemple de lancement problème domination

5.2 Explication code

Pour cette question nous avons créé un package "allPiece" qui va contenir les classes lié a une Piece et un package "question3Choco" qui est le package où se trouve la classe "Main" principale ainsi que la classe "Jeu" qui lance le tout.

Le fonctionnement se passe ainsi, pour n'importe quel type de problème, on va instancier le nombre exact de Piece de type Tour, Cavalier, Fou et Vide. Nous allons mettre chacun de ces objets dans un tableau et on va parcourir ce tableau de telle manière à jouer avec chaque piece. Nous vérifions bien le fait qu'on ne traite pas la même piece avec elle même (auto menacement). Nous avons aussi une contraintes concernant la position de chaque pieces, deux pieces ne peuvent pas avoir la meme position puisqu'elle doivent etre distincte.

Si le problème est une celui de l'indépendance, une piece ne peut pas s'auto menacer et ne peut pas menacer les autres pieces non Vide. Voici le pseudo code :

```
if (i!=k){ //si pas la meme piece
        Piece pieceAttaque=allPiece.get(i); //premiere piece
        Piece pieceSubit=allPiece.get(k); //deuxieme piece

        Constraint unique=pieceAttaque.unique(pieceSubit); //contrainte d'unicite
        unique.post();

        if ((pieceAttaque.getType()!="*") && (pieceSubit.getType()!="*")){
                //si il ne s'agit pas d'une piece qui vide qui attaque une piece vide
                Constraint attaque= model.not(pieceAttaque.Menace(pieceSubit));
                attaque.post();
        }
}
```

Maintenant regardons pour le problème de domination. En plus du fait que c'est la negation de la domination ($\neg\neg x = x$) il faut gérer le cas lorsqu'il y a des obstacles pour les Tours et Fous. Si il y a quelques choses entre la cible et une Tour alors celle-ci ne sera pas menacé. Il s'agit pour ce cas d'un problème d'indépendance pour les Tours et les Fous. Voici le pseudo code :

```
if (i!=k){ //si pas meme piece
        Piece pieceAttaque=allPiece.get(i); //premiere piece
        Piece pieceSubit=allPiece.get(k); //seconde piece

        Constraint unique=pieceAttaque.unique(pieceSubit);
        unique.post(); //position unique

        if ((pieceAttaque.getType()=="*") && (pieceSubit.getType()!="*")){
                //si un case vide alors elle doit etre menace
                Constraint attaque = pieceSubit.Menace(pieceAttaque);
                //case non vide menace une case vide
                OR_contraintes.add(attaque);
        }
        if ((pieceAttaque.getType()=="T") && (pieceSubit.getType()=="F")){
                //gestion des obstacles pour Tour et pour Fou
                Constraint attaque = model.not(pieceAttaque.Menace(pieceSubit));
                OR_contraintes.add(attaque);
        }
}
```

Au niveau des contraintes, on fait un OR de toutes les contraintes dans les deux cas de figures.

5.3 Contrainte Tour

Regardons à present la classe Tour du package allPiece. Les contraintes pour dire qu'une Tour menace une piece sont tres simple. Regadons d'abord le pseudo code :

```
public Constraint Menace(Piece pieceCible){
        Model model=this.getModel();
        Constraint memeLigne=model.arithm(this.getCoordLigne(), "=", pieceCible.getCoordLigne());
        Constraint memeColonne=model.arithm(this.getCoordColonne(), "=", pieceCible.getCoordColonne());
        return model.or(memeLigne,memeColonne);
}
```

En d'autres termes, nous disons tout simplement qu'une piece ne peut pas avoir la meme ligne ou la meme colonne qu'une piece tour si elle veut ne pas etre menacé par celle-ci.

5.4 Contrainte Fou

Regardons la classe Fou du package allPiece pour comprendre comment fonctionne les contraintes pour le type de piece Fou.

```
public Constraint Menace(Piece pieceCible){
 Model model=this.getModel();
 Constraint constraint =model.arithm(this.getCoordColonne().sub(pieceCible.getCoordColonne()).abs().intVar(),
 "=",this.getCoordLigne().sub(pieceCible.getCoordLigne()).abs().intVar());

 return constraint;
}
```

Pour cette contrainte nous nous basons sur ce que le professeur a donné en cours concernant le jeu des 8 reines. Verifier qu'une piece a la meme diagonale qu'une autre revient a faire $|i-k| = |j-l|$ avec (i, j) les coordonne de la premier piece et (k, l) les coordonne de la seconde piece.

5.5 Contrainte Cavalier

Pour ce dernier type de piece, procedons de la meme maniere. Le pseudo code :

```
public Constraint Menace(Piece pieceCible){
        Model model=this.getModel();
        ArrayList<Constraint> cavalierAttaque=new ArrayList<Constraint>();

        //i+1 j+2
        Constraint memeLigne=model.arithm(this.getCoordLigne().add(1).intVar(), "=", pieceCible.getCoordLigne());
        Constraint memeColonne=model.arithm(this.getCoordColonne().add(2).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i+1 j-2
        memeLigne=model.arithm(this.getCoordLigne().add(1).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().sub(2).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i-1 j+2
        memeLigne=model.arithm(this.getCoordLigne().sub(1).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().add(2).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i-1 i-2
        memeLigne=model.arithm(this.getCoordLigne().sub(1).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().sub(2).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i+2 j+1
        memeLigne=model.arithm(this.getCoordLigne().add(2).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().add(1).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i+2 j-1
        memeLigne=model.arithm(this.getCoordLigne().add(2).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().sub(1).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i-2 j+1
        memeLigne=model.arithm(this.getCoordLigne().sub(2).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().add(1).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        //i-2 j-1
        memeLigne=model.arithm(this.getCoordLigne().sub(2).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().sub(1).intVar(), "=", pieceCible.getCoordColonne());
        cavalierAttaque.add(model.and(memeLigne,memeColonne));

        return model.or(cavalierAttaque.toArray(new Constraint[]{}));
}
```

Ce qu'on fait ici c'est tout simplement dire que pour un cavalier, la piece qui est menacé par ce cavalier doit se trouver à la position du saute du cavalier.

5.6 Contrainte Vide

Nous nous sommes rendu compte qu'on a besoin de cette classe juste pour certaintes fonctionne comme l'affichage et comme la verification qu'une piece qui doit attaque ou qui se fait attaquer n'est pas une piece vide. Il n'y a donc aucun intérêt a montrer les contraintes dans ce cas puisqu'il ne renvoi qu'un null, on n'utilise jamais cette fonction menace pour ce type de piece mais puisque nous avons choisi de faire une classe abstraite pour toutes les pieces, il a été nécessaire de devoir redéfinir cette méthode ici.

Ce qui ne faut pas oublier de mentionner c'est que cette classe nous sert également pour les positions. En effet le fait de verifier a chaque fois que chaque piece est distincte oblige l'utilisation des pieces Vides qui ont, elles aussi une position.

5.7 Question Bonus

Pour cette question bonus, il nous a été demander de pouvoir laisser le choix a un utilisateur de définir lui meme un type de piece. Pour cela nous avons simuler un utilisateur qui creer une classe Pion. Cette classe tout comme les autres pieces herite de la classe abstraite Piece et doit definir imperativement la methode : *public Constraint Menace(Piece pieceCible){}*

Voici ce que nous avons defini pour le type de piece Pion :

```
public Constraint Menace(Piece pieceCible){
        Model model=this.getModel();
        ArrayList<Constraint> pionAttaque=new ArrayList<Constraint>();

        //i+1 j+1
        Constraint memeLigne=model.arithm(this.getCoordLigne().add(1).intVar(), "=", pieceCible.getCoordLigne());
        Constraint memeColonne=model.arithm(this.getCoordColonne().add(1).intVar(), "=", pieceCible.getCoordColonne());
        pionAttaque.add(model.and(memeLigne,memeColonne));
        //i+1 j-1
        memeLigne=model.arithm(this.getCoordLigne().add(1).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().sub(1).intVar(), "=", pieceCible.getCoordColonne());
        pionAttaque.add(model.and(memeLigne,memeColonne));

        //i-1 j+1
        memeLigne=model.arithm(this.getCoordLigne().sub(1).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().add(1).intVar(), "=", pieceCible.getCoordColonne());
        pionAttaque.add(model.and(memeLigne,memeColonne));

        //i-1 i-1
        memeLigne=model.arithm(this.getCoordLigne().sub(1).intVar(), "=", pieceCible.getCoordLigne());
        memeColonne=model.arithm(this.getCoordColonne().sub(1).intVar(), "=", pieceCible.getCoordColonne());
        pionAttaque.add(model.and(memeLigne,memeColonne));

        return model.or(pionAttaque.toArray(new Constraint[]{}));
}
```

Au echec le pion peut aller dans une seule direction, nous avons voulu dresser le cas ou c'est un pion qui par dans toutes les directions. Pour pouvoir faire fonctionner ce Pion, l'utilisateur devra modifier le fichier Main.java afin d'ajouter la possibilité de specifier le nombre de Pion et d'utiliser ce nombre dans la classe Jeu.java afin de creer le nombre d'instance necessaire et de les ajouter dans le tableau avec toutes les pieces. Par la suite il pourra tester son programme avec le nouveau type de piece ajouté.

6 Question 4

7 Question 5