

# Projet : INFO-F-302 Informatique Fondamentale.

George Rusu et Maximilien Romain

21 mai 2017

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Question 1</b>	<b>2</b>
2.1	Contraintes tour . . . . .	2
2.2	Contraintes fou . . . . .	2
2.3	Contraintes cavalier : . . . . .	3
<b>3</b>	<b>Question 2</b>	<b>3</b>
3.1	Ensemble Tours . . . . .	3
3.2	Ensemble fous . . . . .	3
3.3	Ensemble cavaliers : . . . . .	4
3.4	Contrainte finale : . . . . .	4
<b>4</b>	<b>Question 3</b>	<b>4</b>
4.1	Lancement du programme . . . . .	4
4.2	Explication code . . . . .	5
4.3	Contrainte Tour . . . . .	5
4.4	Contrainte Fou . . . . .	6
4.5	Contrainte Cavalier . . . . .	6
4.6	Contrainte Vide . . . . .	6
4.7	Question Bonus . . . . .	6
<b>5</b>	<b>Question 4</b>	<b>7</b>
5.1	Contraintes . . . . .	7
5.2	Exemple d'utilisation . . . . .	8
<b>6</b>	<b>Question 5</b>	<b>8</b>
6.1	Contraintes CSP . . . . .	8
6.2	Parser . . . . .	9
6.3	Placements des caméras . . . . .	10
6.4	Contraintes camera . . . . .	10
6.5	Affichage de la solution . . . . .	10
<b>7</b>	<b>Conclusion</b>	<b>10</b>

## 1 Introduction

Le premier objectif de ce projet est de modéliser divers problèmes en problèmes de satisfaction de contraintes (CSP). Le second objectif est d'implémenter un programme résolvant ces problèmes 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 jeu  $V$  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 décisions

**Domaines** :  $D = \{vide, fous, cavalier, tour\}$

**Contraintes** : Pour chacune des pièces de l'échiquier, si une case est occupé par un pions, alors dans la portée de ce pion, donc les cases attaquables par ce pion, doivent être obligatoirement vide. On parcourt 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'échiquier, si une case de cette colonne est occupée par une tour, alors le reste des cases de cette colonne doivent être vide :

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

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

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

### 2.2 Contraintes fou

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

$$\begin{aligned} c_{F,2*n-2} &= ((x_{1,n-1}, x_{2,n}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\}) \wedge \\ &\quad ((x_{1,2}, x_{2,1}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\}) \\ &\quad \vdots \\ c_{F,n+1} &= ((x_{1,2}, x_{2,3}, \dots, x_{n-1,n}), \{(b_1, b_2, \dots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\}) \wedge \\ &\quad ((x_{1,n-1}, x_{2,n-2}, \dots, x_{n-1,1}), \{(b_1, b_2, \dots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\}) \\ c_{F,n} &= ((x_{1,1}, x_{2,2}, \dots, x_{n,n}), \{(b_1, b_2, \dots, b_n) | b_i = F, b_j = V, \forall j \neq i\}) \wedge \\ &\quad ((x_{1,n}, x_{2,n-1}, \dots, x_{n,1}), \{(b_1, b_2, \dots, b_n) | b_i = F, b_j = V, \forall j \neq i\}) \\ c_{F,n-1} &= ((x_{2,1}, x_{3,2}, \dots, x_{n,n-1}), \{(b_1, b_2, \dots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\}) \wedge \\ &\quad ((x_{2,n}, x_{3,n-1}, \dots, x_{n,2}), \{(b_1, b_2, \dots, b_{n-1}) | b_i = F, b_j = V, \forall j \neq i\}) \\ &\quad \vdots \\ c_{F,2} &= ((x_{n-1,1}, x_{n,2}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\}) \wedge \\ &\quad ((x_{n-1,n}, x_{n,n-1}), \{(b_1, b_2) | b_1 = F, b_2 = V \vee b_1 = V, b_2 = F\}) \end{aligned}$$

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

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

### 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  $i + e, j + e, e \in [+1, -1, +2, -2]$

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

## 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 décisions

**Domaines :**  $D = \{Vide, tour, fous, cavalier\}$

**Contraintes :** Pour chacune des pièces de l'échiquier, 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.

### 3.1 Ensemble Tours

Pour chaque case de l'échiquier 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 indiquer 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 être non occupées. Nous avons donc une telle contrainte pour la ligne et la colonne de la case vide menacé par une tour.

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

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

### 3.2 Ensemble fous

Pour chaque case de l'échiquier 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 gêne le fou entre la case vide et ce fou.

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

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

### 3.3 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 vérifier qu'il existe une case dans les 8 cases correspondant à l'ensemble des mouvements du cavalier  $(\{+1, -1, +2, -2\})$ , qui soit occupé par un cavalier.

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

### 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 seul pion menace une case vide, nous pouvons relier nos ensemble par des *unions*  $\cup$  et des *intersections* de contraintes  $\cap$ .

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

## 4 Question 3

Pour cette question nous avons du implementer les deux problèmes décrit ci-dessus. Nous sommes tout d'abord parti sur le fait d'utiliser une matrice de IntVar où chaque case prendrait un numero distinct. Nous nous sommes vite rendu compte que cela n'était pas du tout lisible est très difficile à debugger. Après de nombreuses recherches et de nombreuses questions posées à l'assistant, nous avons eu le déclic que le problème de domination est l'inverse de l'indépendance. En effet, si une pièce menace une autre, il s'agit du problème de domination et inversement pour le problème d'indépendance, une pièce ne doit pas menacer une autre. La negation fait donc son apparition.

Pour pouvoir utiliser ce concept avec la negation, nous avons 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 pièces, on pourrait créer une classe Piece qui représenterai une pièce du Jeu. Mais chaque type de pièce 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.

### 4.1 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'independance
* * 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 pièce Tour (ici 2)
- -f 1 pour le nombre de pièce 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

## 4.2 Explication code

Pour cette question nous avons créé un package "allPiece" qui va contenir les classes lié a une pièce 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 pièce 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 pièce. Nous vérifions bien le fait qu'on ne traite pas la même pièce avec elle même (auto menacement). Nous avons aussi une contraintes concernant la position de chaque pièces, deux pièces ne peuvent pas avoir la meme position puisqu'elle doivent etre distincte.

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

```

if (l!=k){ //si pas la meme piece
    Piece pieceAttaque=allPiece.get(l); //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 ( $\neg x = x$ ) 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 (l!=k){ //si pas meme piece
    Piece pieceAttaque=allPiece.get(l); //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.

## 4.3 Contrainte Tour

Regardons à present la classe Tour du package allPiece. Les contraintes pour dire qu'une Tour menace une pièce 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 pièce ne peut pas avoir la meme ligne ou la meme colonne qu'une pièce tour si elle veut ne pas etre menacé par celle-ci.

## 4.4 Contrainte Fou

Regardons la classe Fou du package allPiece pour comprendre comment fonctionne les contraintes pour le type de pièce 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 pièce a la meme diagonale qu'une autre revient a faire  $|i - k| = |j - l|$  avec  $(i, j)$  les coordonne de la premier pièce et  $(k, l)$  les coordonne de la seconde pièce.

## 4.5 Contrainte Cavalier

Pour ce dernier type de pièce, 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 j-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 pièce qui est menacé par ce cavalier doit se trouver à la position du saute du cavalier.

## 4.6 Contrainte Vide

Nous nous sommes rendu compte qu'on a besoin de cette classe juste pour certaines fonctionne comme l'affichage et comme la verification qu'une pièce qui doit attaque ou qui se fait attaquer n'est pas une pièce 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 pièce mais puisque nous avons choisi de faire une classe abstraite pour toutes les pièces, 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 pièce est distincte oblige l'utilisation des pièces Vides qui ont, elles aussi une position.

## 4.7 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 pièce. Pour cela nous avons simuler un utilisateur qui creer une classe Pion. Cette classe tout comme les autres pièces 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 pièce 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 j-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 pièces. Par la suite il pourra tester son programme avec le nouveau type de pièce ajouté.

## 5 Question 4

Pour cette question nous avons dut implementer une programme qui pour une grille de taille  $n$  va nous donner le nombre minimal de cavalier afin que le problème de domination<sup>1</sup> soit respecté. Nous sommes parti sur une matrice de IntVar de taille  $n * n$  avec pour chaque case un domaine compris entre 0 et 1. Nous avons defini au prealable qu'une case ayant un 0 sera consideré comme vide et inversement une case contenant un 1 sera consideré comme remplie par un et un seul cavalier.

### 5.1 Contraintes

Afin de pouvoir minimiser, nous devons ajouter nos contraintes. Nous pouvons nous aider du fait qu'une case ne peut que soit être vide, soit contenir un cavalier. Nous allons donc pour chaque case parcourir la matrice de IntVar<sup>2</sup> et ajouter les contraintes de la facon suivante :

```
public Constraint menace(int ligne, int colonne){
    ArrayList<Constraint> cavalierAttaque=new ArrayList<Constraint>();

    //i+1 j+2
    if ((ligne+1<this.n) && (colonne+2<this.n)){
        Constraint menace=this.model.arithm(this.echequier[ligne+1][colonne+2], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }
    //i+1 j-2
    if ((ligne+1<this.n) && (colonne-2>=0)){
        Constraint menace=this.model.arithm(this.echequier[ligne+1][colonne-2], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }

    //i-1 j+2
    if ((ligne-1>=0) && (colonne+2<this.n)){
        Constraint menace=this.model.arithm(this.echequier[ligne-1][colonne+2], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }

    //i-1 j-2
    if ((ligne-1>=0) && (colonne-2>=0)){
        Constraint menace=this.model.arithm(this.echequier[ligne-1][colonne-2], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }

    //i+2 j+1
    if ((ligne+2<this.n) && (colonne+1<this.n)){
        Constraint menace=this.model.arithm(this.echequier[ligne+2][colonne+1], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }
    //i+2 j-1
    if ((ligne+2<this.n) && (colonne-1>=0)){
        Constraint menace=this.model.arithm(this.echequier[ligne+2][colonne-1], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }

    //i-2 j+1
    if ((ligne-2>=0) && (colonne+1<this.n)){
        Constraint menace=this.model.arithm(this.echequier[ligne-2][colonne+1], "!=", this.echequier[ligne][colonne]);
        cavalierAttaque.add(menace);
    }
    //i-2 j-1
    if ((ligne-2>=0) && (colonne-1>=0)){
        Constraint menace=this.model.arithm(this.echequier[ligne-2][colonne-1], "!=", this.echequier[ligne][colonne]);
    }
}
```

1. soit une case est occupé soit elle est menacé
2. cette matrice represente la grille que nous devons dominer avec le moins de cavalier possible

```

        cavalierAttaque.add(menace);
    }
    if (!cavalierAttaque.isEmpty()) {
        return this.model.or(cavalierAttaque.toArray(new Constraint[] {}));
    }
    return null;
}

```

Ce que nous faisons ici c'est tout simplement dire que pour chaque case, la case attaquante respective, donc la case où doit se trouver un cavalier doit être différente d'elle-même. En effet nous avons que deux valeurs dans les domaines de chaque case ce qui nous laisse l'opportunité de jouer avec la différence entre la case cible et celle qui attaque.

Voici comment nous ajoutons nos contraintes et comment nous définissons la variable à minimiser :

```

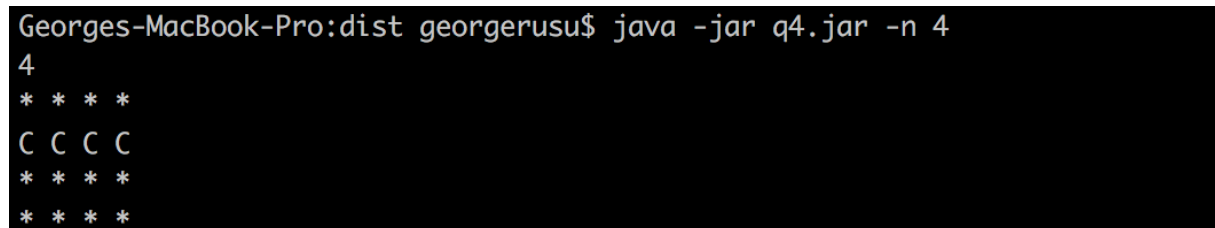
public void MinimizationCavalier() {
    for (int l=0; l<this.n; l++) {
        for (int k=0; k<this.n; k++) {
            if (this.menace(l,k) != null) { //si il existe une contrainte
                this.menace(l,k).post();
            }
            else {
                this.model.arithm(this.echequier[l][k], "=", 1).post(); //sinon c'est une case occupée -> cavalier
                this.sum=this.sum.add(this.echequier[l][k]).intVar();
            }
        }
    }
    this.numCavalier.eq(sum).post();
    this.model.setObjective(Model.MINIMIZE, this.numCavalier);
    [...]
}

```

Puisque à nouveau si une case contient un cavalier, sa valeur sera égale à 1 ou 0 dans le cas d'une case vide, nous ajoutons la contrainte que la somme de toutes les cases doit être égale au nombre de cavalier qu'on doit minimiser.

## 5.2 Exemple d'utilisation

Voici un exemple d'utilisation :



```

Georges-MacBook-Pro:dist georgerusu$ java -jar q4.jar -n 4
4
* * * *
C C C C
* * * *
* * * *

```

FIGURE 3 – Exemple de lancement problème indépendance

où

- -n 4 pour la taille de l'échiquier (ici nous avons défini une taille de 4)

## 6 Question 5

Pour cette question nous avons dû implémenter un programme qui étant donné une grille pouvait calculer le nombre minimal de caméra nécessaire pour que chaque case soit surveillée. Ce problème est identique au problème de domination où chaque case est soit occupée soit surveillée. Pour ce faire nous avons d'abord exprimé ce problème en CSP.

### 6.1 Contraintes CSP

Nous avons un musée qui est représenté par une grille de  $n * m$

**Variables de décision**  $X = \{x_{i,j} | \forall i, j (1 \leq i \leq n). (1 \leq j \leq m)\}$

**Domaines** :  $D = \{0, 1, 2, 3, 4, 5\}$  correspondant respectivement à une case vide, une caméra Nord, une caméra Sud, une caméra Est, une caméra Ouest et un mur.

**Contraintes** : La contrainte du musée est définie comme ceci : chaque case vide doit être surveillée par au moins une caméra. La solution doit être trouvée avec un nombre de caméra minimal.

Pour appliquer la contrainte, il faut donc prendre en compte chaque case du musée. Pour cela nous prenons chaque  $x_{i,j}$  compris dans la taille du musée. Ensuite il faut vérifier que pour chacune de ces cases,



si elle est non occupée, il faut qu'elle soit surveillé par au moins une caméra. Ainsi nous avons une suite d'ensemble de valeurs liées par des relations *OR*, qui nous permet de dire : existe au moins une caméra observant la case vide.

Afin d'être sûr qu'une caméra pointe vers le bon sens, lorsque nous parcourons les lignes et colognes d'une case vide, nous précisons bien qu'il y ai au moins une caméra regardant vers le Nord dans les cases se trouvant en dessous de notre case  $x_{i,j}$ , et ainsi de suite pour les autres directions.

$$C = ((x_{i,j}, \forall i, j (1 \leq i \leq n). (1 \leq j \leq m)), \{b_{i,j}, \forall i, j (1 \leq i \leq n). (1 \leq j \leq m) | [(b_{i,j} = 0) \wedge (\forall k (i < k \leq n), \exists b_{k,j} = 1)] \vee [(b_{i,j} = 0) \wedge (\forall k (0 \leq k < i), \exists b_{k,j} = 2)] \vee [(b_{i,j} = 0) \wedge (\forall k (j < k \leq m), \exists b_{i,k} = 3)] \vee [(b_{i,j} = 0) \wedge (\forall k (0 \leq k < j), \exists b_{i,k} = 4)]\})$$

Afin de réduire le nombre de caméra, nous avons reformulé de la demande d'optimisation. C'est à dire que nous n'allons plus chercher à minimiser le nombre de caméra, mais à maximiser le nombre de case surveillé, ce qui équivaut à la même solution. En effet en réduisant le nombre de caméra, on maximise le nombre de case surveillé avec un petit nombre de caméra.

Il n'y a donc pas de contrainte *CSP* à écrire. Il suffit de donner dire au programme de trouver la solution avec le plus de case vide possible.

## 6.2 Parser

La premiere chose a faire c'est de lancer le programme en lui donnant en parametre un fichier qui contiendra l'architecture du musée. Pour ce faire le programme prend en paramètre d'exécution un fichier texte sous le format suivant :

```
*****
*   ***   *
*         *
*   **  ** *
*   **  ** *
*         ** *
*****
```

Une fois que le programme récupère ce fichier, il va récupérer tout d'abord les dimensions du musée sans compter les quatre mur de la pièce. Une fois terminé, ces valeurs seront donnée aux attributs du programme. Une matrice *grid* est ensuite créée. Cette matrice représente la situation de la pièce du musée avec les murs interieur, et permettra ainsi au programme de placer les caméras aux bons endroits. Voici un exemple de lancement :

```
Georges-MacBook-Pro:dist georgerusu$ java -jar q5.jar -f grid.txt
Parsing File: grid.txt
6
* * * * * * * *
*   * * * E   *
*   E   S     S *
*   * *   * * *
*   * *   * * *
* N E       * * *
* * * * * * * *
```

### 6.3 Placements des caméras

Maintenant que la matrice *grid* à été initialiser, nous pouvons appliquer nos contraintes en fonction de chaque case de cette matrice. Nous faisons alors appel à *minCamera()*, qui va parcourir chaque case de notre matrice. Si la case est différent de "\*", c'est à dire un mur, alors nous allons appliquer nos contraintes. Pour ce faire nous initialisons un tableau qui va contenir toutes nos contraintes, ce tableau est *contrainteTotal\_OR*. C'est ce tableau qui va permettre d'indiquer au solveur, si une case est vide, alors elle est surveillé par au moins une caméra. Ensuite pour la case vide, il va falloir dire : il existe au moins une caméra Sud dans les cases au dessus de notre case vide, ou alors une caméra Nord dans les cases en dessous, ou alors une caméra Est dans les cases à gauche, ou une caméra Ouest dans les cases à droite. Pour ce faire on crée à nouveau un tableau (*existsCam\_OR*) qui va contenir ces contraintes. On parcourt alors en croix les cases adjacentes à la case vide, en prenant soin de nous arrêter lorsque nous rencontrons un mur. Lors de ce parcours, nous ajoutons à notre tableau la contrainte arithmétique : case parcouru = caméra visant notre case vide. Une fois le parcours en croix terminé, il suffit d'ajouter au tableau *contrainteTotal\_OR* quatre contraintes indiquant que la case que nous examinons peut aussi être rempli par une caméra. Il ne reste plus qu'à poster les contraintes du tableau grâce à un *post()* de notre tableau.

### 6.4 Contraintes camera

Pour ce qui contrainte pour une camera, voici un exemple de comment on génère une contrainte pour un type de camera, ici il s'agit pour une camera Nord :

```
[...]
//camera N
m = 1;
while(m < this.nbrLigne){
    if(!this.grid[m][k].equals("*")){
        Constraint contrainte_N = model.arithm(this.salle[m][k], "=", this.NORD);
        existCam_OR.add(contrainte_N);
    }
    else{
        m = 100; //etre sur qu'on va arreter
    }
    ++m;
}
[...]
```

Pour les autres types de camera, c'est le meme principe. On parcourt donc le plan de la salle et si une position n'est pas un mur alors on va ajouter les contraintes de camera : pour chaque position on regarde si il y a des camera Nord, Sud, Est et Ouest et ensuite on fait un OR de tout ca.

Il ne s'agit ici pas en effet de la meilleur approche en programmation et certainement pas de la meilleure optimisation.

Afin de trouver le nombre minimal de camera, nous nous sommes dit que cela revient au meme de maximiser le nombre de case vide et donc de cases surveillé. Nous avons donc une contrainte supplémentaire sur le nombre de cases vides :

```
[...]
ArrayList<BoolVar> tabldimension=new ArrayList<BoolVar>();
for (int i=0;i<this.nbrLigne;i++){
    for (int j=0;j<this.nbrColonne;j++){
        tabldimension.add(this.salle[i][j].eq(this.VIDE).boolVar());
    }
}
BoolVar[] tableau= new BoolVar[this.nbrColonne*this.nbrLigne];
model.sum(tabldimension.toArray(tableau),"=", numV).post();
[...]
```

Pour pouvoir compter le nombre de case vide nous ajoutons une contrainte sur toutes les case : chaque case doit etre vide et on test cette contrainte grace à un boolVar qui renverra true si la condition est respecté. Par la suite il nous suffit de compter le nombre de boolvar dans ce tableau et de l'associer a la variable IntVar *numV* qui represente le nombre de case vide. Il nous suffira donc de maximizer ce nombre.

### 6.5 Affichage de la solution

Pour afficher la solution, il ne reste plus qu'à parcourir le tableau contenant les différentes positions des caméras utilisé par le solveur (*this.salle*). Ce tableau contient la structure de la salle, les cases vides, et le entiers correspondant au caméra et leur sens. Voici un exemple de solution :

## 7 Conclusion

Ce projet nous as permit de mieux comprendre comment Choco-Solveur fonctionne. Nous avons malgré tout eu un grand problème avec le manque de documentation pour les debutant de Choco, comment fonctionne Choco, comment utiliser

```
Georges-MacBook-Pro:dist georgerusu$ java -jar q5.jar -f grid.txt
Parsing File: grid.txt
6
* * * * * * * * *
*   * * * E   *
*   E   S   S *
*   * *   * *   *
*   * *   * *   *
* N E   * *   *
* * * * * * * * *
```