INFO-F-302 Informatique fondamental : Projet : problème de satisfaction de contraintes et utilisation de l'outil ChocoSolver

Rémy Detobel et Denis Hoornaeret

May 21, 2017

Contents

1 Résolution du problème d'échec

1.1 Définition du CSP

Le problème d'indépendance et de domination ont beaucoup de points communs. On peut donc déjà définir plusieurs choses telle que les variables, les domaines ainsi que quelques contraintes.

1.1.1 Mouvement des différentes pièces

Pour résoudre les différents problèmes liés au jeu d'échec, il faut définir les déplacements que peut faire un pion. Dans les figures ci-dessous, la case grise représente la position d'un pion et les cases vertes représentent les cases où le pion peut se rendre. Lors de la résolution de nos problèmes, il faudra faire attention à une règle importante du jeu d'échec. Cette règle stipule qu'une pièce a se trouvant sur une des cases atteignables par une pièce b, va rendre les cases derrière elle non atteignables par la pièce b (dans la figure ci-dessous, ces cases sont rouges). La seule exception à la règle est le mouvement du cavalier comme montré dans la figure 3.

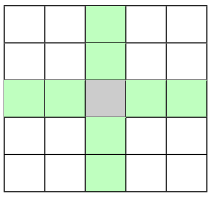

Figure 1: Tour

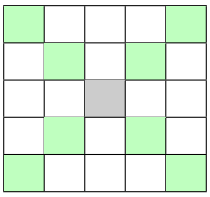

Figure 2: Fou

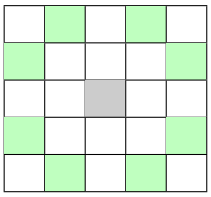

Figure 3: Cavalier

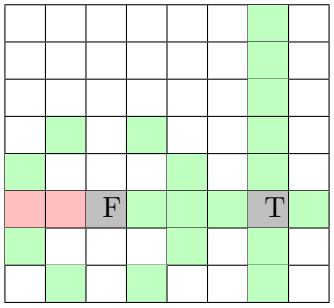

Figure 4: Règle de masquage

1.1.2 Ensemble de variables :

Pour ces deux problèmes, on va modéliser l'échiquier comme étant une matrice de variables. L'échiquier étant carré, la matrice le sera également.

$$X = \{x_{ij} \ \forall \ 0 \leq i, j < n\}$$

Où :

- n est la taille de l'échiquier;
- x_{ij} est une case de cet échiquier.

1.1.3 Ensemble de domaines :

Ici, toutes les variables ont le même domaine qui correspond à:

$$D = \{D_x \mid x \in X\}$$

$$D_x = \{C, T, F, V\}$$

Où :

- C est un cavalier;
- T est une tour;
- F est un fou;
- V est une case vide.

1.1.4 Ensemble des contraintes :

Contrainte de position : On définit des contraintes pour chaque cases. En fonction du problème, la case devra contenir ou non un pion et devra menacer ou non d'autres case. Quoi qu'il en soit, on définit pour chaque case les contraintes suivantes:

$$C = \{C_{x_{i,j}} \ \forall \ x_{i,j} \in X\}$$

$$C_{x_{i,j}} = \{CavMvt(x_{i,j}) \ \vee \ TourMvt(x_{i,j}) \ \vee \ FouMvt(x_{i,j}) \ \vee \ VideMvt(x_{i,j})\}$$

Où i, j sont compris entre 0 et n et où "$TypeMvt$" est une fonction permettant de faire des vérifications en fonction des mouvements d'une pièce d'un certain type. Ce seront ces fonctions qui seront définies dans la suite de ce document.

Contrainte du nombre de pion Le problème de domination et le problème d'indépendance définissent tous les deux un certain nombre de pions k à placer sur le plateau. Pour vérifier que ce nombre de pion est correcte, on peut définir cette contrainte de la manière suivante :

$$\forall k \mid \sum_{x_{i,j} \in X} x_{i,j} = k$$

1.2 Question 1: Problème d'indépendance

Pour rappel, le problème d'indépendance consiste à trouver une disposition où aucune pièce n'est menacée par une autre. Ce qui revient à dire, que pour toute pièce disposée sur l'échiquier, il n'y a que des cases vides dans l'ensemble des cases atteignables par la pièce en question.
Les variables et le domaine ont déjà été défini au point 1.1.2 et 1.1.3. Les contraintes ont également déjà été définie dans le point 1.1.4. Il reste encore à décrire les fonctions permettant de définir pour une case donnée le rôle qu'elle devra avoir en fonction de la pièce.

1.2.1 Définition des fonctions de mouvement :

Comme indiqué dans le point précédent on va vérifier que chaque case est vide ou contient un pion qui n'attaque aucune autre case. Les fonctions suivantes permettent donc de faire cette vérification.

Tour : La fonction ($TourMvt(x_{i,j})$) ci-dessous vérifie que pour tout mouvement d'une tour sur l'axe verticale et horizontale, aucune case ne soit occupée. L'ensemble des zones atteintes est mis en évidence dans la figure 1.

$$\forall \ 0 \leq k < n \mid (k \neq i \ \wedge \ k \neq j \ \wedge \ x_{i,j} = T \ \wedge \ x_{k,j} = V \ \wedge \ x_{i,k} = V)$$

Fou : Cette fonction ($FouMvt(x_{i,j})$) vérifie, comme la fonction précédente qu'aucune case atteignable ne soit occupé par un pion. Dans le cas d'un fou, il faut donc vérifier sur ses deux diagonales. L'ensemble des zones atteintes est mis en évidence dans la figure 2.

$$x_{i,j} = F \ \wedge$$

$$(\forall k \ (0 \leq i-k < i) \ \wedge \ (0 \leq j-k < j) \ x_{i-k,j-k} = V \ \wedge$$
$$\forall k \ (i < i+k < n) \ \wedge \ (0 \leq j-k < j) \ x_{i+k,j-k} = V \ \wedge$$
$$\forall k \ (0 \leq i-k < i) \ \wedge \ (j < j+k < n) \ x_{i-k,j+k} = V \ \wedge$$
$$\forall k \ (i < i+k < n) \ \wedge \ (j < j+k < n) \ x_{i+k,j+k} = V)$$

Cavalier : La fonction ($CavMvt(x_{i,j})$) ci-après vérifie également que le déplacement d'un cavalier depuis la case $x_{i,j}$ ne menace pas une autre pièce. Le mouvement du cavalier est un peu particulier car c'est la seule pièce qui peut "traverser" d'autres pièces. L'ensemble des zones atteintes est mis en évidence dans la figure 3.

$$x_{i,j} = C \ \wedge$$

$$\forall k_1, k_2 \in \{-1, -2, 1, 2\} \mid (|k_1| \neq |k_2|) \ \wedge \ (0 \leq i+k_1 < n) \ \wedge \ (0 \leq i+k_2 < n) \ \wedge \ x_{i+k_1,j+k_2} = V$$

Vide : Si une case est vide, elle est bien évidemment considérée comme correcte. On a donc la fonction $VideMvt(x_{i,j})$ tel que :

$$x_{i,j} = V$$

1.3 Question 2: Problème de domination

Pour rappel, le problème de domination consiste à trouver une disposition où toutes les pièces sont menacées par au moins une autre pièce. Ce qui revient à dire, que pour toute pièce disposée sur l'échiquier, il n'y a aucune case qui n'est pas occupée ou menacée par une autre. Comme pour le problème d'indépendance, les variables et le domaine sont définis en point 1.1.2 et 1.1.3. Il en est de même pour les contraintes où il ne reste plus qu'à définir les fonctions permettant de définir le rôle que devra avoir une case donnée.

1.3.1 Définition des fonctions de mouvement :

Tour : La fonction ($TourMvt(x_{i,j})$) ci-dessous décrit le mouvement d'une tour sur l'axe verticale et horizontale. L'ensemble des zones atteintes est mis en évidence dans la figure 1.

$$\exists k \ (0 \leq k < n)$$

$$(\forall t \ (i < t \leq k) \ \vee \ (k \leq t < i) \mid x_{k,j} = T \ \wedge \ (x_{t,j} = V \ \vee \ t = k)) \ \vee$$
$$(\forall t \ (j < t \leq k) \ \vee \ (k \leq t < j) \mid x_{i,k} = T \ \wedge \ (x_{i,t} = V \ \vee \ t = k)$$

Fou : Pour décrire le mouvement d'un fou sur ces deux diagonales, on définit la méthode $FouMvt(x_{i,j})$:

$$[(\exists k \mid (0 \leq i-k < i) \ \wedge \ (0 \leq j-k < j) \ x_{i-k,j-k} = F \ \wedge$$

$$\forall t \mid (0 \leq t < i-k) \ \wedge \ (0 \leq t < j-k) \ \wedge \ x_{i-t,j-t} = V) \ \vee$$
$$(\exists k \mid (0 \leq i+k < i) \ \wedge \ (0 \leq j-k < j) \ x_{i+k,j-k} = F \ \wedge$$
$$\forall t \mid (0 \leq t < i+k) \ \wedge \ (0 \leq t < j-k) \ \wedge \ x_{i+t,j-t} = V)$$

$$(\exists k \mid (0 \leq i-k < i) \ \wedge \ (0 \leq j+k < j) \ x_{i-k,j+k} = F \ \wedge$$
$$\forall t \mid (0 \leq t < i-k) \ \wedge \ (0 \leq t < j+k) \ \wedge \ x_{i-t,j+t} = V)$$
$$(\exists k \mid (0 \leq i+k < i) \ \wedge \ (0 \leq j+k < j) \ x_{i+k,j+k} = F \ \wedge$$
$$\forall t \mid (0 \leq t < i+k) \ \wedge \ (0 \leq t < j+k) \ \wedge \ x_{i+t,j+t} = V)]$$

Cavalier : Contrairement aux déplacements précédent, la fonction vérifiant les déplacements du cavalier ($CavMvt(x_{i,j})$) ne doit pas vérifier qu'une pièce gêne un potentiel déplacement. La figure 3 montre les déplacements possibles pour cette pièce.

$$\exists k_1, k_2 \in \{-1, -2, 1, 2\} \mid (|k_1| \neq |k_2|) \ \wedge \ (0 \leq i+k_1 < n) \ \wedge \ (0 \leq i+k_2 < n) \ \vee \ x_{i+k_1,j+k_2} = C$$

Vide : Les cases vides ne sont pas acceptées si elles ne sont pas attaquées. Cependant, on peut voir le problème dans l'autre sens et donne à la fonction "$VideMvt$" le rôle de vérifier que la case actuelle est différente du vide. Si tel est le cas, elle sera quoi qu'il arrive valide. On peut donc écrire:

$$x_{i,j} \neq V$$

1.4 Question 3: Implémentation informatique et résultats

Dans la section présente, les différents exemples présents dans l'énoncé seront reproduits et la quantité de solutions possibles pour ces exemples sera donnée.
En plus des paramètres demandés dans l'énoncé, il est possible de passer d'autres options au démarrage de l'application :

-h permet d'afficher tous les paramètres pouvant être passé en paramètre au démarrage de l'application;

-all permet d'afficher tous les résultats;

-utf8 permet d'afficher la(les) solution(s) sous une forme plus agréable à l'oeil;

-debug montre quelques informations qui aident à voir ce qu'il se passe;

-time permet de voir le temps d'exécution du programme;

opti permet d'indiquer pour un pion donné (`-c`, `-f` ou `-t`) qu'on aimerait que le programme minimise le nombre de fois que cette pièce apparait.

1.4.1 Résultats du problème d'indépendance

Pour obtenir les résultats de l'exemple du problème d'indépendance, qui contient un fou, un cavalier et une tour sur un échiquier de trois cases de largeur, la commande à taper est la suivante :

```
java -jar Echiquier.jar -i -n 3 -f 1 -c 1 -t 1
```

Il est possible que le résultat affiché ne corresponde pas à celui présent dans l'énnoncé. Pour remédier à cela, on peut demander au programme d'afficher toutes les solutions via la commande suivante:

```
java -jar Echiquier.jar -i -n 3 -f 1 -c 1 -t 1 -all
```

Parmis les 16 résultats obtenus, on retrouve bien celui présenté dans l'exemple.

```
***
**T
FC*
```

Remarques :

Beaucoup de réponses pourraient être considérées comme équivalantes car la disposition des pièces sur l'échiquier est la même à une ou deux rotations près.

*** **T FC*	*T* **C **F	*CF T** ***	F** C** *T*

Un autre exemple était également présent dans l'énoncé. Voici un résultat équivalent (pour avoir la même réponse que celle présentée, il faut afficher tous les résultats en rajoutant le paramètre `-all`).

```
java -jar Echiquier.jar -i -n 4 -t 2 -f 1 -c 2
**T*
*T**
C***
F**C
```

On peut également remarquer qu'en cas de problème insoluble comme celui suggéré par la commande ci-après, le programme affiche bien un message indiquant qu'aucun résultat n'est possible:

```
java -jar Echiquier.jar -i -n 4 -t 5
pas de solution
```

1.4.2 Résultats du problème de domination

Pour obtenir les résultats de l'exemple du problème de domination, qui contenait un fou, un cavalier et deux tours sur un échiquier de trois cases de largeur, la commande à taper est la suivante :

```
java -jar Echiquier.jar -d -n 3 -f 1 -c 1 -t 2 -all
```

Parmi les 348 résultats obtenus, on retrouve bien celui présenté dans l'exemple:

```
T*F
*T*
**C
```

1.4.3 Détail technique

Maven est utilisé pour faciliter la compilation du projet. Il intègre automatiquement ChocoSolver et l'inclut même dans l'un des deux fichiers exécutable produit (seul celui finissant par "with-dependencies" contient les sources de ChocoSolver).

1.5 Question Bonus: Création d'un pion personnalisé

Toutes les classes permettant de gérer des pions sont appelés "Manager". En effet, cette classe va "manager" les pions d'un certain type pour les positionner correctement.

Création de la classe Pour implémenter son propre pion, il faut créer une classe qui devra hériter soit de `PionManager`, soit de `OpaquePionManager`. La différence entre les deux se situe simplement dans le fait qu'un pion "opaque" ne peut pas traverser les autres pions. Par exemple une tour est un pion opaque contrairement au cavalier.

La classe qui permettra de gérer ce nouveau pion devra définir une méthode et un constructeur. Pour les classes "opaques", une méthode en plus devra être intégrée.

Canvas :
Le canvas de base pour créer une classe est le suivant:

```
public class TestManager extends OpaquePionManager {

    public TestManager(NbrPions nbrPion, int tailleEchec) {
        super("nomPion", nbrPion, tailleEchec, 'S', '8');
    }

    @Override
    public ArrayList<Integer[]> getAccessibleCase(int currentLigne,
     int currentColonne) {
        // Code
    }

    /**
     * Uniquement pour les cases opaques
     */
    @Override
    protected ArrayList<Integer[]> getEmptyCase(int currentLigne,
     int currentColonne, int currentDecalageLigne, int currentDecalageColonne) {
        // Code
    }

}
```

Le constructeur doit appeler le constructeur parent en précisant le nom du pion (le paramètre `"nomPion"`) (uniquement utilisé pour du débug) ainsi que le symbole qui sera utilisé pour les résultats lors de l'affichage classique ou UTF-8 (respectivement `'S'` et `'8'` dans cet exemple).

La méthode `getAccessibleCase` permet de spécifier toutes les cases accessibles par ce pion depuis les coordonnées passées en paramètre, à savoir `(currentLigne, currentColonne)`. Cette méthode retourne donc une liste de coordonnées relatives aux coordonnées passées en paramètre.

La méthode `getEmptyCase` ne doit pas être définie que pour les pions "opaques". Cette méthode prend 4 paramètres en entré. Ces derniers sont respectivement: la position sur la ligne du pion que l'on observe, la position sur la colonne du pion que l'on observe, le déplacement du pion sur la ligne et le déplacement du pion sur la colonne. En d'autres mots, ces paramètres peuvent être lus deux par deux. Les deux premiers définissent les coordonnéees du pion actuel : `(currentLigne, currentColonne)`. Les deux suivants définissent le potentiel déplacement

du pion: `(currentDecalageLigne, currentDecalageColonne)`. La méthode `getEmptyCase` retourne la liste des cases qui doivent être vides entre `(currentLigne, currentColonne)` et `(currentDecalageLigne, currentDecalageColonne)` pour que le pion puisse être déplacé.

Déployer Une fois la classe crée, il faut indiquer au programme que celle-ci existe. Pour cela, il suffit de modifier la classe `PionManager` et d'instancier la classe qui vient d'être créé dans la méthode `initAllManager`.

Lecture des entrées Par défaut, le programme ne saura pas combien de pions il doit placer. Il lit cette information directement dans les arguments utilisés pour lancer le programme. Pour la classe venant d'être créé, rien n'a été implémenté pour permettre de directement définir le nombre de pions devant être placé. Il faut donc soit modifié la classe `ReadingArguments` pour palier à cela, soit fixé cette valeur dans le programme (dans `initAllManager` ou directement dans le constructeur du nouveau pion) mais il faudra alors recompiler le projet à chaque modification de cette valeur.

1.6 Question 4: Minimisation du nombre de cavalier

Le problème est réalisable grâce au code développé pour la question 3. Il suffit de minimiser le nombre de cavalier présent dans la matrice de variables. Pour cela, ChocoSolver met à disposition la méthode `model.among`. Cette méthode permet de compter le nombre de numéro présent dans une liste de variables. Dans le cas présent on lui indiquera le numéro représentant le cavalier en demandant au solver de minimiser cette variable passée en paramètre à la méthode `among`.
Pour exécuter l'exemple présenté dans l'énoncé, il faut taper l'instruction suivante :

```
java -jar Echiquier.jar -d -n 4 -c opti -all
```

Le programme permet d'obtenir exactement la solution proposée dans l'énoncé :

```
****
*CC*
*CC*
****
```

En plus de cette réponse, il en existe d'autre : 9 au total.

> Remarque :
>
> Le programme proposé est assez flexible que pour proposer une résolution d'une optimisation pour toutes les pièces possibles dans tous les types de problèmes possibles (de domination et d'indépendance).

2 Résolution de la surveillance de musée

Le problème de surveillance de musée consiste à positionner un ensemble minimal de caméras dont la zone de surveillance est fixe de manière à ne laisser aucune zones sans surveillance.

2.1 Définition du CSP

2.1.1 Ensemble de variables :

Pour ce problème, on va modéliser le plan du musée comme étant une matrice de variables. Le musée n'étant pas forcement carré, la matrice aura une largeur de m et une hauteur de n.

$$X = \{x_{ij} \ \forall \ i(0 \leq i < n) \ \forall \ i(0 \leq j < m)\}$$

Où :

- n est la hauteur du plan du musée;
- m est la largeur du plan du musée;
- x_{ij} est une case du plan du musée.

2.1.2 Ensemble de domaines :

Ici aussi toutes les variables ont le même domaine qui correspond à:

$$D = \{D_x \mid x \in X\}$$

$$D_x = \{M, V, N, S, E, O\}$$

Où :

- M est un mur;
- V est une case vide;
- N est un capteur Nord;
- S est un capteur Sud;
- E est un capteur Est;
- O est un capteur Ouest.

2.1.3 Ensemble des contraintes :

Pour chaque case on définit la contrainte suivante:

$$C = \{C_{x_{i,j}} \ \forall \ x_{i,j} \in X\}$$

$$C_{x_{i,j}} = \{CheckMur(x_{i,j}) \wedge [CameraN(x_{i,j}) \vee CameraS(x_{i,j}) \vee CameraE(x_{i,j}) \vee CameraO(x_{i,j})]\}$$

Une case du plan n'est valide que s'il s'agit d'un mur ou qu'elle est visible depuis une caméra. La première fonction "$CheckMur$" vérifie si la case actuelle est un mur. Les autres fonctions, qui commencent par "Caméra" vérifient qu'un certain type de caméra peut bien voir la case actuelle.

Murs La fonction "CheckMur" se base sur une fonction $isAWall(x)$ renvoyant `true` ($\top$) si x est inclus dans cet ensemble et `false` ($\bot$) sinon. En d'autres mots, elle pourra être définie comme ceci:

$$isAWall(x) = x \in Walls$$

Où :

- $Walls$ est l'ensemble des variables définies comme étant des murs.

Typiquement, cette fonction sert à émuler un système de variables constantes. Le contenu de l'ensemble $Walls$ est défini après le *parsing* du fichier. On va donc pouvoir définir la fonction $checkMur(x_{i,j})$ comme étant égal à:

$$(isAWall(x_{i,j}) \ \wedge \ x = M) \ \vee \ (\neg isAWall(x) \ \wedge \ x \neq M)$$

Cette contrainte permet donc de vérifier qu'une variable ne deviendra pas un mur et que les murs ne se "transformeront" pas en autre chose. Il aurait également été possible de modifier le domaine de ces cases mais cela aurait compliqué la mise en place des autres contraintes.

> Remarque :
>
> La caméra d'un type donné ne peux pas voir une caméra de son type car cela signifierait qu'il y a redondance.

Les 4 points suivant définissent chacun une fonction de type "$CameraX$" où X est le type de caméra (respectivement N, S, O et E).

Nord

$$\exists k \ (0 \leq k < i)$$

$$\forall t \ (k \leq t < i) \mid (x_{k,j} = N \ \wedge \ (x_{t,j} = V \ \vee \ t = k))$$

Sud

$$\exists k \ (i < k < n)$$

$$\forall t \ (i < t \leq k) \mid (x_{k,j} = S \ \wedge \ (x_{t,j} = V \ \vee \ t = k))$$

Ouest

$$\exists k \ (j < k < m)$$

$$\forall t \ (j < t \leq k) \mid (x_{i,k} = O \ \wedge \ (x_{i,t} = V \ \vee \ t = k))$$

Est

$$\exists k \ (0 \leq k < j)$$

$$\forall t \ (k \leq t < j) \mid (x_{i,k} = E \ \wedge \ (x_{i,t} = V \ \vee \ t = k))$$

2.2 Implémentation et résultat

La commande permettant d'exécuter le programme de surveillance est la suivante :

```
java -jar Surveillance.jar -file exemple.txt -all
```

2.2.1 Détails sur l'implémentation

Pour démarrer le programme, il faut lui spécifier le fichier à démarrer. Cela doit être fait via les paramètres passés au programme lors de son démarrage.

-all permet d'afficher l'ensemble des résultats;

-utf8 permet d'afficher la(les) solution(s) sous une forme plus agréable à l'oeil;

-debug montre plusieurs informations qui aident à voir ce qu'il se passe;

-file permet de spécifier le chemin vers le fichier qu'on désire lire;

-gui permet d'ouvrir une boite de dialogue permettant de sélectionner un fichier via un gui;

-time affiche le temps total d'exécution;

-fulltime permet d'afficher régulièrement le temps d'exécution (au moins tous les 20 secondes).

2.2.2 Résultats

Voici la résolution de l'exemple proposé dans l'énoncé :

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

```
6
*********
* ***E  *
*      O*
* **S** *
*N** ** *
*  O **N*
*********
```

Au total ce problème possède 480 solutions (le programme met plus de 2 minutes 30 pour tout calculer). Voici un autre exemple de résolution :

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

```
5
*********
*S*******
*     OS*
* ** ** *
* **N** *
*   O** *
*********
```

Qui, lui, peut être résolu en moins de 20 secondes et qui compte au total 168 solutions différentes.

2.2.3 Détail technique

Un fichier maven est utilisé pour compiler le projet. Celui-ci nécessite la version complète de Java intégrant JavaFX pour pouvoir être compilé. Comme pour la première partie de ce projet, ChocoSolver est automatiquement ajouté dans l'exécutable contenant "with-dependencies".